Energy-Aware Real-Time Scheduling in Embedded Multiprocessor Systems

Thèse présentée en vue de l’obtention du grade de Docteur en Sciences

Directeur de thèse : Joël GOOSSENS
Co-directeur : Dragomir MILOJ EVIC

Vincent NELIS
Année académique 2010–2011
Acknowledgements

Writing a Ph.D thesis is a long and difficult task that requires much concentration and patience. As my family and friends can attest, I spent the last six months cooped up in my apartment, declining almost all invitations to go out and socialize. But this long drafting phase represents only the tip of the iceberg, because this book is primarily the result of painstaking and exciting research, which would have been impossible without the help of several key people. Thereby, I would like to thank them here for their support and I apologize in advance for the ones I might forget.

Foremost, I would like to thank my supervisor, Prof. Joël Goossens, whose method and attitude towards research often served as a model for myself. I am very grateful to him not only for giving me the opportunity to experience such an adventure, but also for convincing me to follow this Ph.D curriculum, for his permanent support and his steady confidence in my work. Also, I would like to thank him for giving me so much freedom and encouraging me to do my own choices. I definitely learned a lot that way. His continuous guidance, his thorough reviews and his helpful suggestions led almost all our papers to be accepted.

I am also so grateful to Prof. Dragomir Milojevic for his continuous guidance during the past two years. During my four years as a PhD-student, I had the privilege of working in two successful departments, that both provided a pleasant and friendly environment. Many thanks to all the people in the Department of Computer Science at Université Libre de Bruxelles, where I started on the graduate studies path. In particular, I am very grateful to Vandy Berten and Patrick Meumeu Yomsi for their advices and the nice discussions we had. My best regards also go to the people in the Embedded Electronics Group of BEAMS Department at Université Libre de Bruxelles, and especially to Geoffrey Nelissen with who I spent so many hours dealing with tortuous problems.

I am grateful to the Fonds National de la Recherche Scientifique of Belgium for funding this research.
My gratitude also goes to all the colleagues with who I shared so pleasant working times, namely, the Professors Raymond Devillers, Shelby Funk, Gerhard Fohler, Isabelle Puaut, Nicolas Navet, Lilana Cucu-GrosJean, and the Ph.D Björn Andersson, Linh Thi Xuan Phan, Damien Hardy, Mike Holenderski and Raphaël Guerra.

My family and all my friends have played an important role throughout my studies and I profit from this opportunity to thank each of them. To Benjamin Willems, Jimmy Ackermans, Bruno Cheval, Pascal Van der straeten, Mohamed ben Yaacoub, Lionel Brouhier, Mehdi Felloussia, Antoine Marot, Gabriel Kalyon, Olivia Antonello, Celia Capuzzo, Laura Zadunayski, Flavia Najih, Ouarda Djedidene, Florence Verbeurght and to every one that I am not citing here but that I don’t forget. Last but not least, special thanks to Nathalie Santoromito for her helpful advices and her continuous support throughout my studies.

Finally, on the more personal side, I would like to express my thanks to my parents and my brother Olivier Nélis, for their blind trust in me and for encouraging me always to do my own choices, and to Vanessa Merolla, for her patience, her support and for taking such good care of me. I don’t know if it is the right time to say that, but in my opinion there is no wrong time. I love you all.
Author’s awards

2006 – Solvay Award

My master thesis has been awarded by Solvay Company, an international chemical and pharmaceutical Group with headquarters in Brussels. Employing about 29,000 people in 50 countries, Solvay is listed on the Euronext stock exchange in Brussels and is one of 20 world leaders in chemistry.

2010 – Best Paper Award

Our paper entitled “Scheduling Multi-Mode Real-Time Systems upon Uniform Multi-processor Platforms” has been awarded in Bilbao (Spain) during the 15th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2010).
Author’s publication list

Papers in Preparation


Refereed Conference and Workshop Papers


▶ Vincent Nelis, Joël Goossens and Björn Andersson. Two Protocols for Scheduling Multi-Mode Real-Time Systems upon Identical Multiprocessor Platforms. In 21st Eu-


Research Reports


**Theses**


« The most exciting phrase to hear in science, the one that heralds new discoveries, is not “Eureka!” but rather, “hmm.... that’s funny...”.

[La phrase la plus excitante à entendre en science, celle qui annonce de nouvelles découvertes, n’est pas “Eurêka !”, mais plutôt “Tiens, c’est marrant...”.]

Isaac Asimov
Table of contents

1 Introduction to embedded real-time systems 1
  1.1 Overview of an embedded system .............................. 3
  1.1.1 Definitions and examples of application ................... 3
  1.1.2 Constraints and notion of real-time ....................... 6
  1.1.3 Typical design of embedded computer systems ............... 8
  1.2 The application layer ....................................... 10
  1.2.1 Notion of real-time application .......................... 10
  1.2.2 Task models ............................................. 12
  1.2.3 Multimode application model .............................. 20
  1.3 Real-Time Operating Systems (RTOS) ......................... 22
  1.3.1 Overview of the main functionalities ...................... 22
  1.3.2 Memory management ...................................... 23
  1.3.3 Interrupt handlers ....................................... 25
  1.3.4 Inter-task communication and resource sharing ............. 26
  1.3.5 Real-time schedulers .................................... 29
  1.4 The physical layer ........................................... 40
  1.4.1 Overview of the main components .......................... 40
  1.4.2 The different steps for manufacturing an integrated circuit 40
  1.4.3 Processors in embedded systems .......................... 44
  1.4.4 Power dissipation of a processor .......................... 45
## TABLE OF CONTENTS

1.4.5 ASIC vs. FPGA implementation ........................................ 55  
1.4.6 Models of multiprocessor architectures ............................... 56  
1.5 Energy consumption of embedded real-time systems .................. 58  
1.5.1 Context of the problem ................................................ 58  
1.5.2 Straightforward solutions to reduce the energy consumption .... 58  
1.5.3 The Dynamic Voltage and Frequency Scaling (DVFS) framework 60  
1.6 Outline of the thesis .................................................. 61  

2 Scheduling Multimode Real-Time Applications .......................... 71  
2.1 Introduction .................................................................. 73  
2.1.1 Motivation ............................................................... 73  
2.1.2 Problematic of multimode real-time applications ................ 74  
2.1.3 Related work .......................................................... 75  
2.1.4 Contribution and organization ....................................... 76  
2.2 Models of computation and specifications ............................... 77  
2.2.1 Application and platform model ..................................... 77  
2.2.2 Mode transition specifications ....................................... 79  
2.2.3 Scheduler specifications ............................................. 81  
2.3 The synchronous protocol SM-MSO .................................... 85  
2.3.1 Description of the protocol .......................................... 85  
2.3.2 Main idea of the validity test ....................................... 87  
2.4 The asynchronous protocol AM-MSO .................................. 90  
2.4.1 Description of the protocol .......................................... 90  
2.4.2 Main idea of the validity test ....................................... 93  
2.5 Some preliminary results for determining validity tests ............. 100  
2.5.1 Introduction to the three required key results .................... 100  
2.5.2 Demonstration of the first key result ............................... 101  
2.5.3 Demonstration of the second key result ........................... 103  
2.5.4 Presentation of some base results ................................... 109
TABLE OF CONTENTS

2.5.5 Organization for the third key result .................................. 113
2.6  Validity test for identical platforms and FJP schedulers ............ 115
2.6.1 Determination of upper-bounds on the idle-instants ............... 115
2.6.2 Determination of a validity test ....................................... 124
2.7  Validity test for identical platforms and FTP schedulers .......... 125
2.7.1 Determination of upper-bounds on the idle-instants ............... 125
2.7.2 Determination of a validity test ....................................... 128
2.8  Validity test for uniform platforms and FJP schedulers .......... 129
2.8.1 Some useful observations .............................................. 129
2.8.2 Determination of upper-bounds on the idle-instants ............... 134
2.8.3 Determination of a validity test ....................................... 137
2.8.4 Another analysis of the maximum makespan: the main idea ..... 138
2.8.5 A second upper-bound on the makespan ............................ 145
2.8.6 A third upper-bound on the makespan .............................. 163
2.9  Validity test for uniform platforms and FTP schedulers .......... 166
2.9.1 Determination of upper-bounds on the idle-instants ............... 166
2.9.2 Determination of a validity test ....................................... 170
2.10 Adaptation to identical platforms of the upper-bound for uniform platforms 171
2.11 Accuracy of the proposed upper-bounds ............................... 177
2.12 Simulation results .......................................................... 187
2.13 Validity tests at a glance .................................................. 191
2.14 Conclusion and future work ............................................... 195

3  Optimizing the hardware design ........................................ 199
3.1  Context of the problem .................................................... 201
3.1.1 Motivation ................................................................. 201
3.1.2 Related works ............................................................ 203
3.1.3 Chapter organization ..................................................... 203
3.2  Model of computation ...................................................... 204
# TABLE OF CONTENTS

3.2.1 Hardware model ........................................... 204
3.2.2 Application model ........................................... 206
3.2.3 Scheduling specifications ................................. 209

3.3 Offline task mapping algorithms ............................... 211
3.3.1 Our methodology ............................................ 211
3.3.2 Notations ..................................................... 214
3.3.3 Schedulability test for global-DM ......................... 214
3.3.4 Schedulability test for global-EDF ....................... 215
3.3.5 Power dissipation model of the MPSoC platforms ....... 218
3.3.6 Formulation of the problem .............................. 220
3.3.7 Approximation algorithms ............................... 221

3.4 Simulation results ............................................. 226

3.5 How to handle mode changes? ................................. 230

3.6 Conclusion and future work .................................. 235

4 Exploiting the DVFS framework .................................... 243

4.1 Introduction ................................................... 245

4.2 Model of computation and assumptions ....................... 246
4.2.1 Platform model ............................................ 246
4.2.2 Application model ........................................... 248
4.2.3 Scheduling specifications ................................. 249

4.3 Related work .................................................. 250
4.3.1 Uniprocessor energy-aware algorithms ................... 250
4.3.2 Multiprocessor energy-aware algorithms ............... 254
4.3.3 Energy-aware algorithms with other concerns .......... 256

4.4 Contribution and organization of the chapter ............... 258

4.5 Offline speed determination .................................... 259
4.5.1 Determination of an identical speed: a generic approach .. 261
4.5.2 Determination of an identical speed: a method specific to EDF . 268
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5.3</td>
<td>Determination of individual processor speeds</td>
<td>274</td>
</tr>
<tr>
<td>4.6</td>
<td>Online slack reclaiming algorithms</td>
<td>277</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Notion of slack time</td>
<td>277</td>
</tr>
<tr>
<td>4.6.2</td>
<td>The Multiprocessor Online Reclaiming Algorithm (MORA)</td>
<td>281</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Multiprocessor One Task Extension (MOTE)</td>
<td>297</td>
</tr>
<tr>
<td>4.6.4</td>
<td>Combination MORA – MOTE</td>
<td>312</td>
</tr>
<tr>
<td>4.7</td>
<td>Simulation results</td>
<td>314</td>
</tr>
<tr>
<td>4.7.1</td>
<td>The process to generate our applications</td>
<td>314</td>
</tr>
<tr>
<td>4.7.2</td>
<td>The simulated methods</td>
<td>317</td>
</tr>
<tr>
<td>4.7.3</td>
<td>Results provided by our offline methods</td>
<td>320</td>
</tr>
<tr>
<td>4.7.4</td>
<td>Results provided by our online methods</td>
<td>322</td>
</tr>
<tr>
<td>4.8</td>
<td>Conclusion</td>
<td>331</td>
</tr>
</tbody>
</table>

Conclusions and perspectives 342

Appendix 347

A. Processors characteristics 347
B. Additional experiments for our DVFS methods 350
C. Missing proofs of Section 2.8.6 351

Table of symbols 379
List of Figures

1.1 Satellites are typical examples of embedded technology .......................... 5
1.2 Common structure of a computer system. Systems are generally structured in three layers: the application layer, the Operating System (OS) and the hardware layer. ................................................................. 9
1.3 A fictive example where an intermediate controller is used in order to regulate the parameters of the real-time process. .............................. 11
1.4 Schedule of $\tau_i = (2, 3, 4, 6)$, where $\tau_i = (\text{offset}, \text{WCET}, \text{deadline}, \text{period})$. ........................................ 15
1.5 A sprinkler system installed in a 3-floor building. ................................. 19
1.6 A possible execution scenario of the sprinkler system on a single-processor architecture. ................................................................. 20
1.7 Illustration of a multimode real-time application composed of 5 operating modes. The arrows that link the modes represent the mode transitions that can occur at run-time. Here, mode transitions are possible between every mode. ................................................................. 20
1.8 Memory management based on fixed-size-blocks algorithm. .................... 24
1.9 Because most of the existing schedulability analyses do not take overheads into account, a job can sometimes miss its deadline even if the system was asserted to be schedulable in theory. ................................. 31
1.10 Global vs. Partitioned approaches. ......................................................... 38
1.11 Schedule of $\tau_1 = (2, 3)$, $\tau_2 = (4, 6)$ and $\tau_3 = (6, 12)$ (where $\tau_i = (C_i, D_i = T_i)$) on a 2-processors platform with the global FTP assignment: $\tau_1 > \tau_2 > \tau_3$. ......................................................... 39
1.12 Scheme of a simple CMOS input-inverter. ............................................ 49
1.13 Illustration of the behavior of an ideal CMOS inverter. ............................. 50
1.14 The through current flows from source to ground when both transistors are conducting. .................................................. 51

2.1 Thanks to our definition of weakly work-conserving schedulers, only one schedule is possible for a given set of jobs, an identical platform and a specific priority assignment. .................................................. 84

2.2 For any fixed set of jobs and any uniform platform, the schedule generated by any strongly work-conserving scheduler forms a staircase. This phenomenon will be sometimes referred as the “staircase property” in the remainder of this chapter. .................................................. 85

2.3 Illustration of a mode transition handled by SM-MSO. ................. 86

2.4 Illustration of a mode transition handled by AM-MSO. ................. 91

2.5 Illustration of the idle-instants. ........................................... 95

2.6 With FJP schedulers, multiple job priority assignments can be derived from the same worst-case rem-job set. .......................... 98

2.7 During the time interval \([\text{idle}_3(J, \pi, P), \text{idle}_3(J', \pi, P)]\) 3 jobs are running in \(S'\) while only 2 jobs are running in \(S\). .................. 104

2.8 During the time interval \([\text{idle}_3(J, \pi, P), \text{idle}_3(J', \pi, P)]\) 3 jobs are running in \(S'\) while only 2 jobs are running in \(S\). .................. 105

2.9 Bijection between the sets \(J_{wc,i}^a\) and \(J_{any}\) ......................... 108

2.10 Within the time interval \([\text{idle}_3, \text{idle}_4]\), the tasks in \(\tau(1) \cup \tau(2) \cup \tau(3) \cup \tau(4)\) benefit from 4 processors in the actual schedule (Figure 2.10b) while only 3 processors are available to these tasks in the schedule assumed by Algorithm 4 (Figure 2.10a). ................................. 111

2.11 Chart representation of the contributions presented in this chapter, where the labels (a), (b), (c) and (d) illustrate the relation between the contributions: (a) the upper-bound \(\overline{ms}_{1}^{\text{unif}}(J, \pi)\) is more pessimistic than \(\overline{ms}_{1}^{\text{ident}}(J, \pi)\), (b) the upper-bound \(\overline{ms}_{2}^{\text{unif}}(J, \pi)\) is a generalization of \(\overline{ms}_{1}^{\text{ident}}(J, \pi)\), (c) we conjecture that the upper-bound \(\overline{ms}_{3}^{\text{unif}}(J, \pi)\) is more pessimistic than \(\overline{ms}_{1}^{\text{ident}}(J, \pi)\) and (d) the computation of \(\overline{ms}_{1}^{\text{unif}}(J, \pi, P)\) and \(\overline{ms}_{1}^{\text{ident}}(J, \pi, P)\) are very different for FTP schedulers, especially because of the difference in the definitions of weakly and strongly work-conserving schedulers. 114

2.12 Illustration of the notion of processed work \(Work_i\) .......................... 126
2.13 Counter-example proving that SJF policy does not always lead to the maximum makespan. ................................. 130

2.14 Upon uniform platforms, on the contrary to identical platforms, if the number of jobs is equal to the number of processors (i.e., \( n = m \)) then the makespan depends on the job priority assignment. ................................. 131

2.15 Illustration showing that Expression 2.15 cannot be naively extended to uniform platforms. Notice that we voluntarily exaggerated the error in this picture since the actual error is \( 20 - 19.9 = 0.1 \) ........................................... 133

2.16 Example of schedule in which the makespan is larger than that returned by \( \overline{ms}^{\text{unif}}_0(J, \pi) \) ......................................................... 139

2.17 Illustration of the time-instant \( t_i \) (Figure 2.17a) and the early completion of job \( J_i \) (Figure 2.17b) .............................................................. 141

2.18 Impossibility for job \( J_i \) to complete later than \( t_i + \frac{c_i}{s_m} \) while being executed in the green areas .................................................. 142

2.19 The amount of execution units in the green area is a lower-bound on the amount of execution units that can be executed within \([\text{idle}^i_1, \text{idle}^i_m]\) because (1) \( \text{idle}^i_1 \) is an upper-bound on \( \text{idle}^i_j \) and (2) \( s_1 \) is (one of) the slowest speed(s) of \( \pi \) ......................................................... 150

2.20 Staircase defined by the \( \text{idle}^{j-1}_i \) ................................................................. 169

2.21 Chart representation of the contributions presented in this chapter, where the labels (a), (b), (c) and (d) illustrate the relation between the contributions: (a) the upper-bound \( \overline{ms}^{\text{unif}}_1(J, \pi) \) is more pessimistic than \( \overline{ms}^{\text{ident}}(J, \pi) \), (b) the upper-bound \( \overline{ms}^{\text{unif}}_2(J, \pi) \) is a generalization of \( \overline{ms}^{\text{ident}}(J, \pi) \), (c) we conjecture that the upper-bound \( \overline{ms}^{\text{unif}}_3(J, \pi) \) is more pessimistic than \( \overline{ms}^{\text{ident}}(J, \pi) \) and (d) the computation of \( \overline{ms}^{\text{unif}}_1(J, \pi, P) \) and \( \overline{ms}^{\text{ident}}(J, \pi, P) \) are very different for FTP schedulers, especially because of the difference in the definitions of weakly and strongly work-conserving schedulers. 172

2.22 Illustration of the equality \( \sum_{k=1}^{m-1} \sum_{i=1}^{n-m+k} c_i = (m - 1) \cdot \sum_{i=1}^{n-m} c_i + \sum_{i=1}^{m-1} (m - i) \cdot c_{n-m+i} \) .............................................. 174

2.23 Illustration of different schedules proving that the upper-bounds provided by Expression 2.12 can be reached. ........................................... 180

2.24 Illustration of a schedule in which the makespan is over-approximated. 181
LIST OF FIGURES

2.25 Simulation results. .......................................................... 190

3.1 Our considered DMP model with \(2 \times (5 \times 5)\) processors. .......... 205

3.2 Illustration of a multimode real-time application composed of 5 operating modes (the same as in Figure 1.7 on page 20). Here, a task mapping is determined beforehand for each mode. That is, the set of tasks of each mode is split into two subsets \(\tau^{lp}\) and \(\tau^{hp}\) and each subset is scheduled upon its respective platform \(L_{lp}\) or \(L_{hp}\) using only a sufficient number of CPUs. The squares depicted in green are the supplied CPUs while the CPUs turned off are represented by crossed-out squares. .......... 213

3.3 Illustration of the crossover operation. Basically, the value of \(k\) divides the two “parent” solutions \(V_1\) and \(V_2\) into two pieces. The first “child” solution \(v_1\) is composed of the first piece of \(V_1\) and the second piece of \(V_2\) whereas the second “child” \(v_2\) is composed of the first piece of \(V_2\) and the second piece of \(V_1\). .......................................................... 223

3.4 Our simulation results for both Global-DM and Global-EDF. ............ 228

3.5 This illustration shows how mode transitions are handled by our synchronous protocol SM-MSO (described in Chapter 2, page 85), using the task mapping approach proposed in this chapter. .............. 232

4.1 Schedule of \(\tau_1 = (2, 4, 5)\), where \(\tau_i = (WCET, deadline, period)\). According to our definition of the speed of a processor, the execution time of job \(\tau_{1,3}\) is twice as long as that of \(\tau_{1,1}\) and \(\tau_{1,2}\) since its execution speed is twice as slow. .......................................................... 249

4.2 Illustration of our approach for the identical speed determination problem.265

4.3 The main idea behind \(EDF^{(k)}\) ............................................. 270

4.4 Distribution of the number of cycles to decode different kinds of video, ranging from news streaming to complex 3D animations. X-axis: number of cycles, y-axis: probability. .......................................................... 279

4.5 Illustration of the worst-case and actual schedules. The internal slack time is depicted in red and the external slack time is depicted in green. .... 280

4.6 Representation of the internal and external slack time. ..................... 282

4.7 Offline and actual schedules. ............................................. 285
4.8 Rules 4.1 and 4.2 of MORA. .................................................. 289

4.9 At time $t$, MOTE detects the unavoidable external slack time after the execution of $\tau_{i,j}$ (represented by the green area) and it slows down the execution speed of $\tau_{i,j}$ accordingly. .................................................. 298

4.10 Reducing an execution can result in a deadline miss. ......................... 303

4.11 Reducing an execution speed must be done very carefully. ............... 304

4.12 Illustration of the functions $\sum_{\tau_{i,j} \in \tau_{i,j}} PotAct_{k}(0, t')$, $\sum_{\tau_{i,j} \in \tau} PotRel_{k}(0, t')$ and $\Pi_{3}(0, t')$, $\forall t' \in [0, a_{3,1} + T_{3}]$. ................................................. 306

4.13 Comparison between the consumption generated by the three methods $I-EDF^{\text{max}}$ (in red), $I-EDF$ (in blue), $I-EDF^{(k)}$ (in green) and the method CLV (in black), using the consumption model of the processor StrongARM SA-1100. The Y-axis gives the consumption of each method, relative to the consumption of CLV which is considered as 100 %. The characteristics of this processor are given in Table 4.8 (page 348). ...................... 321

4.14 Comparison between the consumption generated by the three methods $I-EDF^{\text{max}}$ (in red), $I-EDF$ (in blue), $I-EDF^{(k)}$ (in green) and the method CLV (in black), using the consumption model of the processor StrongARM SA-1100. The Y-axis gives the consumption of each method, relative to the consumption of CLV which is considered as 100 %. The parameter $D_{\text{max}}$ is set to 0.6 and 0.5 in the upper and lower picture, respectively. ........... 323

4.15 Some statistics about the consumption of $I-EDF^{\text{max}}$ (in red), $I-EDF$ (in blue) and $I-EDF^{(k)}$ (in green). In both figures, the X-axis displays every value of $D_{\text{max}}$ while the Y-axis gives the relative consumption of each method (relative to the consumption of CLV). ...................... 324

4.16 Comparison between the consumption generated by the four methods $I-EDF^{\text{max}}$ (in red), MORA-EDF (in light green), MOTE-EDF (in dark green), MORAOTE-EDF (in gold) and the method CLV, using the consumption model of the processor StrongARM SA-1100. The Y-axis gives the consumption of each method, relative to the consumption of CLV which is considered as 100 %. The parameter $D_{\text{max}}$ is set to 0.6 and 0.5 in the upper and lower picture, respectively. ...................... 326
4.17 Some statistics about the consumption of $I_{EDF}^{max}$ (in red), $I_{EDF}(k)$ (in glowing green), $MORA_{EDF}(k)$ (in white), $MOTE_{EDF}(k)$ (in gray) and $MORAOTE_{EDF}(k)$ (in gold). In both figures, the X-axis displays every value of $D_{max}$ while the Y-axis gives the relative consumption of each method (relative to the consumption of CLV).

4.18 This picture represents the average energy savings generated exclusively by the methods $MORA_{EDF}(k)$ (in white), $MOTE_{EDF}(k)$ (in gray) and $MORAOTE_{EDF}(k)$ (in gold), compared to the average energy savings due to the method $I_{EDF}(k)$. In this figure, we consider the consumption model of the processor StrongARM SA-1100 whose characteristics are given in Table 4.8 (page 348).

4.19 Evolution of the ratio $\frac{m}{n}$ (displayed on the Y-axis) for different value of the parameter $D_{max}$.

4.20 Comparison of the consumption profiles of each processor model presented in this Appendix.

4.21 Representation of the different notations used in the proof.

4.22 Some statistics about the consumption of $I_{EDF}^{max}$ (in red), $I_{EDF}$ (in blue) and $I_{EDF}(k)$ (in green), simulated on Transmeta Crusoe TM5400 processors. In both figures, the X-axis displays every value of $D_{max}$ while the Y-axis gives the relative consumption of each method (relative to the consumption of CLV).

4.23 Some statistics about the consumption of $I_{EDF}^{max}$ (in red), $I_{EDF}(k)$ (in glowing green), $MORA_{EDF}(k)$ (in white), $MOTE_{EDF}(k)$ (in gray) and $MORAOTE_{EDF}(k)$ (in gold), simulated on Transmeta Crusoe TM5400 processors. In both figures, the X-axis displays every value of $D_{max}$ while the Y-axis gives the relative consumption of each method (relative to the consumption of CLV).

4.24 Some statistics about the consumption of $I_{EDF}^{max}$ (in red), $I_{EDF}$ (in blue) and $I_{EDF}(k)$ (in green), simulated on PowerPC-405LP processors. In both figures, the X-axis displays every value of $D_{max}$ while the Y-axis gives the relative consumption of each method (relative to the consumption of CLV).
4.25 Some statistics about the consumption of $I_{\text{EDF}}^{\text{max}}$ (in red), $I_{\text{EDF}}^{(k)}$ (in glowing green), $MORA_{\text{EDF}}^{(k)}$ (in white), $MOTE_{\text{EDF}}^{(k)}$ (in gray) and $MORAOTE_{\text{EDF}}^{(k)}$ (in gold), simulated on PowerPC-405LP processors. In both figures, the X-axis displays every value of $D_{\text{max}}$ while the Y-axis gives the relative consumption of each method (relative to the consumption of CLV). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 374

4.26 Some statistics about the consumption of $I_{\text{EDF}}^{\text{max}}$ (in red), $I_{\text{EDF}}$ (in blue) and $I_{\text{EDF}}^{(k)}$ (in green), simulated on Intel PXA270 processors. In both figures, the X-axis displays every value of $D_{\text{max}}$ while the Y-axis gives the relative consumption of each method (relative to the consumption of CLV). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 375

4.27 Some statistics about the consumption of $I_{\text{EDF}}^{\text{max}}$ (in red), $I_{\text{EDF}}^{(k)}$ (in glowing green), $MORA_{\text{EDF}}^{(k)}$ (in white), $MOTE_{\text{EDF}}^{(k)}$ (in gray) and $MORAOTE_{\text{EDF}}^{(k)}$ (in gold), simulated on Intel PXA270 processors. In both figures, the X-axis displays every value of $D_{\text{max}}$ while the Y-axis gives the relative consumption of each method (relative to the consumption of CLV). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 376

4.28 Some statistics about the consumption of $I_{\text{EDF}}^{\text{max}}$ (in red), $I_{\text{EDF}}$ (in blue) and $I_{\text{EDF}}^{(k)}$ (in green), simulated on Intel XScale processors. In both figures, the X-axis displays every value of $D_{\text{max}}$ while the Y-axis gives the relative consumption of each method (relative to the consumption of CLV). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 377

4.29 Some statistics about the consumption of $I_{\text{EDF}}^{\text{max}}$ (in red), $I_{\text{EDF}}^{(k)}$ (in glowing green), $MORA_{\text{EDF}}^{(k)}$ (in white), $MOTE_{\text{EDF}}^{(k)}$ (in gray) and $MORAOTE_{\text{EDF}}^{(k)}$ (in gold), simulated on Intel XScale processors. In both figures, the X-axis displays every value of $D_{\text{max}}$ while the Y-axis gives the relative consumption of each method (relative to the consumption of CLV). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 378
List of Tables

2.1 Processing times of the 12 jobs in $J$. ........................................... 179
2.2 Competitive factors of the presented upper-bounds on the maximum makespan. ................................................................. 187
2.3 Processing times of the 10 jobs in $J$. ........................................... 187
2.4 Statistics issued from the simulation .............................................. 191
3.1 Description of the processor cores .............................................. 227
4.1 Characteristics of the Intel XScale processor family. In [1], the power characteristics are (in mW) 80, 170, 400, 900 and 1600 at frequency 150, 400, 600, 800 and 1000, respectively. The idle power is 40 mW. ............ 347
4.2 Characteristics of the processor PowerPC-405LP. In [4], the power characteristics are (in mW) 19, 72, 600 and 750 at frequency 33, 100, 266 and 333, respectively. The idle power is 12 mW. ................................. 347
4.3 Characteristics of the processor Transmeta Crusoe TM5400 [3]. ....... 348
4.4 Characteristics of the processor StrongARM SA-1100 [5]. ............... 348
4.5 Characteristics of the processor Intel PXA270. From Table 5-7 in [2], the power characteristics are (in mW) 44.2, 116, 279, 375, 570, 747 and 925 at frequency 13, 104, 208, 312, 416, 520 and 624 respectively. The idle power is 8.5 mW. ................................................................. 349
CHAPTER 1

Introduction to Embedded Real-Time Systems

On fait la science avec des faits, comme on fait une maison avec des pierres : mais une accumulation de faits n’est pas plus une science qu’un tas de pierres n’est une maison.

Henri Poincaré

Contents

1.1 Overview of an embedded system ........................................ 3
1.2 The application layer .................................................. 10
1.3 Real-Time Operating Systems (RTOS) ................................. 22
1.4 The physical layer ..................................................... 40
1.5 Energy consumption of embedded real-time systems ............ 58
1.6 Outline of the thesis ................................................... 61
CHAPTER 1. INTRODUCTION TO EMBEDDED REAL-TIME SYSTEMS

Abstract

Nowadays, computer systems are everywhere. From simple portable devices such as watches and MP3 players to large stationary installations that control nuclear power plants, computer systems are now present in all aspects of our modern and every-day life. In about only 70 years, they have completely perturbed our way of life and they reached a so high degree of sophistication that they will be soon capable of driving our cars and cleaning our houses without any human intervention. As computer systems gain in responsibilities, it becomes essential that they provide both safety and reliability. Indeed, a failure in systems such as the anti-lock braking system (ABS) in cars could threaten human lives and generate catastrophic and irreversible consequences. Hence, for many years, researchers have addressed these emerging problems of system safety and reliability which come along with this fulgurant evolution.

This chapter provides a general overview of embedded real-time computer systems, i.e., a particular kind of computer system whose number grows daily. We provide the reader with some preliminary knowledge and a good understanding of the concepts that underlie this emerging technology. We focus especially on the theoretical problems related to the real-time issue and briefly summarize the main solutions, together with their advantages and drawbacks. This brings the reader through all the conceptual layers constituting a computer system, from the software level—the logical part—that specifies both the system behavior and requirements to the hardware level—the physical part—that actually performs the expected treatments and reacts to the environment. In the meanwhile, we introduce the theoretical models that allow researchers for theoretical analyses which ensure that all the system requirements are fulfilled. Finally, we address the energy consumption problem in embedded systems. We describe the various factors of power dissipation in modern technologies and we introduce different solutions to reduce this consumption.
1.1 Overview of an embedded system

1.1.1 Definitions and examples of application

Most computer systems share the same features and/or electronic components and the term “embedded system” is not rigorously defined in the literature. In this thesis, we go along with the definition proposed in [12, 40]: an embedded system is an autonomous microprocessor-based system including both electronics and software and designed to control a function or a range of functions. According to this definition, even though laptops share some elements with embedded systems (such as the operating systems and microprocessors), they do not belong to the “embedded” category since they are designed to support a very large range of end-user needs. On the other hand, even though air traffic control systems involve numerous workstations and networks between airports and radar sites, they may usefully be considered as embedded systems. In this document, we consider that the two key characteristics of an embedded system are:

1. being dedicated to specific tasks,
2. being subject to specific constraints such as energy consumption, size, weight, thermal dissipation, etc.

The fact that an embedded system is dedicated to a finite range of functionalities offers a prior knowledge that influences the whole process of thinking, designing, implementing and manufacturing the system. This knowledge enables such systems to be designed so that their performance closely matches the expected requirements, while benefiting from a high reliability and a reduced cost, size, weight, energy consumption, etc., depending on their field of application. Notice that embedded systems are not only standalone devices, but can consist of a small part of a larger general purpose system. For example, an embedded system in an automobile (e.g., the Anti-lock Braking System ABS) provides a specific function and is a subsystem of the car itself.

Recently, the notion of “Cyber-Physical System” (CPS) has emerged. A CPS is a system featuring a tight combination of, and coordination between, the computational and physical elements of the system. Even though CPS are often referred to as “embedded systems”, embedded systems emphasize more on the computational elements, and less on an intense link between the computational and physical elements. Unlike more traditional embedded systems, a CPS is typically designed as a network
CHAPTER 1. INTRODUCTION TO EMBEDDED REAL-TIME SYSTEMS

of interacting elements instead of as standalone devices [45]. The expectation is that in the coming years, the link between computational and physical elements will be improved by ongoing advances in science and engineering, dramatically increasing the adaptability, autonomy, efficiency, functionality, reliability, safety, and usability of CPS. The advances will broaden the potential of CPS in several dimensions, including: precision (e.g., robotic surgery and nano-level manufacturing), coordination (e.g., air traffic control, war fighting), intervention (e.g., collision avoidance), operation in dangerous or inaccessible environments (e.g., search and rescue, firefighting, and deep-sea exploration), efficiency (e.g., zero-net energy buildings) and augmentation of human capabilities (e.g., healthcare monitoring and delivery) [32]. From year 2006, the term “CPS” has become more and more employed in the literature, where authors often use the terms “cyber-physical systems” and “embedded systems” without making any distinction between them. Out of habit, we will use the traditional term of “embedded system” in the remainder of this document.

Nowadays, embedded systems control many devices in common use, spanning all aspects of our modern life (see Chapter 1 of [13] for details). Physically, they range from portable devices such as digital watches and MP3 players to large stationary installations such as traffic lights, factory controllers, satellites (see Figure 1.1), or the systems controlling nuclear power plants. Complexity varies from low, with a single microcontroller chip, to very high with multiple units, peripherals and networks mounted inside a large enclosure. Among the application fields of the embedded technology, one can cite:

- **Telecommunications systems.** They use numerous embedded systems, from switches in the telephone network to the end-user mobile phones.

- **Consumer electronics.** Among these applications, one can cite:
  - *convenience appliances* which include printers, GPS receivers, DVD players, digital cameras, videogame consoles, mobile phones, MP3 players and Personal Digital Assistants (PDAs).
  - *household appliances*, such as microwave ovens and washing machines, which include embedded systems in order to provide flexibility, efficiency and other features.
  - *home appliances* which use embedded devices in order to control lights, climate, security, audio/visual, surveillance, etc.
1.1 Overview of an embedded system

Figure 1.1: Satellites are typical examples of embedded technology
– advanced HVAC systems (Heating, Ventilating, and Air Conditioning) which use networked thermostats to more accurately control the temperature.

- Transportation systems. From cars to airplanes, transportation systems increasingly use embedded technology. For instance, most of current automobiles provide safety systems such as Anti-lock Braking System (ABS), Electronic Stability Control (ESC/ESP), Traction Control System (TCS) and automatic four-wheel drive. More recently, automobiles such as electric and hybrid vehicles also use embedded systems to maximize efficiency and reduce pollution.

- Medical equipments. They are increasingly developed with more embedded systems for vital signs monitoring, electronic stethoscopes for amplifying sounds, and various medical imaging (PET, SPECT, CT, MRI) for non-invasive internal inspections.

1.1.2 Constraints and notion of real-time

As mentioned in the introduction, reliability has become a key feature of computer embedded systems. Such systems are often incorporated into machines that are expected to run continuously for years without errors, and in some cases recover by themselves if an error occurs. Consequently, the software is generally developed and tested more carefully than that for personal computers, and unreliable mechanical moving parts such as disk drives, switches or buttons are avoided. Among the specific reliability issues, one can cite:

1. The system cannot safely be shut down for repair, or it is too inaccessible to repair. Examples of such systems include space systems, undersea cables, navigational beacons, bore-hole systems, and automobiles.

2. The system must be kept running for safety reasons. Examples include aircraft navigation, reactor control systems, safety-critical chemical factory controls, train signals, engines on single-engine aircraft.

3. The system will lose large amounts of money when shut down: telephone switches, factory controls, bridge and elevator controls, funds transfer and market making, automated sales and service.

Additionally to the guarantee of reliability, embedded systems are often subject to hard constraints, including size and weight limitations, or constraints on thermal
1.1 Overview of an embedded system

dissipation and/or energy consumption. By contrast, systems that are not considered as embedded, such as personal computers for instance, are designed to be flexible and to meet a wide range of end-user needs. Besides, most embedded systems are subject to timing constraints following which the correctness of an operation depends not only upon its logical correctness, but also upon the time in which the operation is achieved. Embedded systems that are subject to such timing constraints are named embedded real-time systems and can be broken into three broad categories: hard real-time, soft real-time and firm real-time systems.

**Hard real-time systems.** The classical conception is that in a hard real-time system, the completion of an operation after its deadline is considered useless—ultimately, this may cause a critical failure of the system or expose end-users to hazardous situations. A simple example of a hard real-time system is, again, the anti-lock braking system on a car—the real-time constraint in this system is the short time in which the brakes must be released to prevent the wheel from locking. Hard real-time systems can be said to have failed if they do not complete a function before its deadline, where the deadlines are always relative to a specific event. In short, in hard real-time systems, the deadline of every operation must be met, regardless of the system load. In practice, hard real-time systems are used when it is imperative to react to an event within a strict deadline. Such strong guarantees are required by systems for which not reacting in a certain interval of time would cause large loss in some manner, especially damaging the surroundings physically or threatening human lives (although the strict definition is simply that missing the deadline constitutes failure of the system). For example, a car engine control system is a hard real-time system because a delayed signal may cause engine failure or damage. Other examples of hard real-time embedded systems include medical systems such as heart pacemakers and industrial process controllers. In embedded systems, hard real-time systems typically interact with the hardware at a low level.

**Soft real-time systems.** The notion of soft real-time is opposite to that of hard real-time. Indeed, even though the completion of an operation before its deadline is still preferred, soft real-time systems are not subject to the timing constraints, i.e., even if all the deadlines are missed, the system can continue to operate. Such systems are nonetheless referred to as “real-time” since they use real-time mechanisms (such as real-time operating systems for instance) in order to meet as many deadlines as possible.
CHAPTER 1. INTRODUCTION TO EMBEDDED REAL-TIME SYSTEMS

Firm real-time systems. Between soft real-time and hard real-time, there is an intermediate paradigm known as firm real-time. In contrast to hard real-time systems, firm real-time systems tolerate some “lateness”, i.e., a deadline miss results only in a decreased quality of service (e.g., omitting frames while displaying a video). Basically, the notion of firm real-time is less strict than that of hard real-time since it allows deadlines to be missed, but it is more strict than soft real-time in the sense that only a predefined ratio of deadline miss is allowed. Firm real-time systems are typically used where there are some issues of concurrent access and the need to keep a number of connected systems up to date with changing situations; for example software that maintains and updates the flight plans for commercial airliners. The flight plans must be kept reasonably current but can operate to a latency of seconds. Live audio-video systems are also usually firm real-time; missing a deadline results in degraded quality, but the system can continue to operate.

In the remainder of this document, we will consider only hard real-time systems. Notice that real-time computing is sometimes misunderstood to be high-performance computing, but this is not always true. For example, a massive supercomputer executing a scientific simulation may offer impressive performance, yet it is not executing a real-time computation. On the other hand, once the hardware and software of an ABS has been designed to meet its deadlines, no further performance gains is necessary. In conclusion, the most important requirement of real-time systems is to offer predictability rather than performance.

1.1.3 Typical design of embedded computer systems

Like many computer systems, most of embedded real-time systems are conceptually structured in three distinct layers as depicted by the chart in Figure 1.2: the application layer, the real-time Operating System and the hardware layer.

The application layer. The application layer contains all the functionalities that the system must provide—the functions that fulfill the requirements imposed by the system designer. Each of these functionalities is implemented either as a real-time or non-real-time process, according to these requirements. Recall that a real-time process is a process subject to “real-time constraints”, i.e., operational deadlines from event to system response, whereas a non-real-time process has no deadline, even if fast response
1.1 Overview of an embedded system

![Diagram of computer system structure]

**Figure 1.2:** Common structure of a computer system. Systems are generally structured in three layers: the application layer, the Operating System (OS) and the hardware layer.

or high performance is desired or preferred. Note that in practice, systems often contain both real-time and non-real-time processes. In the next section, we will describe the theoretical application models that allow researchers for theoretical analyses.

**The Operating System.** Basically, an Operating System (OS) is a software that provides an interface between the hardware and the other softwares. Its main role is to act as a host for the application layer, to manage the user processes and to coordinate the allocations and sharing of the hardware resources. The OS allows the application layer to manipulate the hardware via OS routines which are gathered in interfaces named “Application Programming Interfaces (APIs)”. By invoking these routines, any process of the application layer can request the OS and ultimately interact with some elements of the hardware, such as memory and other external devices (including scanners, printers, etc.). Section 1.3 describes the principal functionalities of the OS and highlights the main differences between real-time and non real-time OS.

**The hardware layer.** The hardware layer is the physical part of the system. It contains interconnected devices such as processors and memory. These interconnections can be as simple as point-to-point connections or can be organized in complex networks. In Section 1.4, we first provide the reader with the multiple hardware models used in
the scheduling theory. Then, since this thesis focuses on the energy consumption, we provide a detailed explanation of how typical CMOS logic gates consume energy.

1.2 The application layer

1.2.1 Notion of real-time application

As introduced in the previous sections, an application is said to be “real-time” when it is subject to timing deadlines from event to system response. Deadlines are usually set by the system designer, but they typically reflect a need for safety or sustainability of the system performances. When deadlines ensure a certain level of safety, such as in ABS applications in cars, they are set up in accordance with the related regulation and are never updated after the system manufacture. For instance, the deadline of an ABS functionality will be set to 5 ms if 5 ms has been defined as an acceptable delay in the regulation of automotive market. Since missing such a deadline could threaten human lives and/or generate catastrophic and irreversible consequences, ABS applications typically fall in the category of hard real-time systems, for which a particular attention is given to the strict respect of the deadlines. Typically, hard real-time systems work in closed and highly predictable environments.

On the other hand, if deadlines allow only to maintain a certain Quality-of-Service (QoS), deadline miss are tolerated and these systems are referred to as soft (or firm) real-time systems. On the contrary to hard real-time systems, soft real-time applications execute in open and unpredictable environments. Examples of soft real-time systems include online trading, e-commerce, and multimedia. Performance guarantees are required in these types of applications because failure to meet performance guarantees may result in loss of customers, financial damage, or liability violations. For such unpredictable environments, adaptive real-time systems have been developed as a promising approach to achieve performance guarantees. While research on hard real-time computing are concerned only with guaranteeing complete avoidance of undesirable effects such as overload and deadline misses, adaptive real-time systems are designed to handle such effects dynamically. An abundant literature (see [1, 2, 3, 4, 5, 19, 20, 22, 51, 59, 60, 63] for instance) is available on adaptive real-time computing, where adaptive real-time devices are often implemented via an intermediate mechanism that regulates the task parameters (deadlines, periods, etc.) at run-time by periodically sending a feedback to
1.2 The application layer

the application. For example, such intermediate mechanism can be found in systems where some processes have to execute an operation at a constant rate, such as sampling a sensor or decoding a video frame. Figure 1.3 depicts a fictive example of such a system where a real-time process (noted \( P \)) running on the computer has to decode and display the pictures sent by the webcam. We assume that the webcam captures and send pictures with a rate of 25 pictures/second. Also, we assume that the process \( P \) has to decode and display 25 pictures/second. More precisely, \( P \) must have decoded and displayed every picture sent by the webcam before the next picture is available. Therefore, its processing deadline is \( \frac{1}{25} = 0.04 \) second for each picture. Since the decoding time depends on the picture, and because the processing time of the other processes running on the computer may also vary over time, it can be the case where the execution rate of 25 pictures/second of \( P \) is too high for the computation capability of the computer. That is, the entire system could be overloaded and some processes could miss their respective deadlines. For this reason, an external controller is installed between the computer and the webcam. Its function is to regulate the rate at which the webcam captures and send the pictures to the computer, so that it can decrease the execution rate of \( P \) (thus increasing its deadline). This enables the system to recover from overload, but implies a reduction of the system QoS, i.e., the number of pictures displayed on the screen every second is reduced.

Figure 1.3: A fictive example where an intermediate controller is used in order to regulate the parameters of the real-time process.
CHAPTER 1. INTRODUCTION TO EMBEDDED REAL-TIME SYSTEMS

Recall that even though systems may run both real-time and non-real-time processes, we focus in this thesis on systems that run only hard real-time applications. In order to conduct theoretical studies, the application needs to be modeled so that (i) properties can be extracted and (ii) schedulability\(^1\) can be verified. Hence, in the next section, we introduce the notations related to the applications that will be used throughout the thesis.

1.2.2 Task models

A process or functionality of the system can be seen as a specific treatment such as decoding a data frames while displaying a video, reading a data from a sensor, etc. Each functionality is modeled by a real-time task \(\tau_i\) which is characterized by a few parameters whose number, role and interpretation obviously depend on the nature of the functionality. From now on, we will say that a task \(\tau_i\) releases a job \(\tau_{i,j}\) (where \(j\) is the index of the job) at time \(t\) to express the fact that \(\tau_i\) is instantiated exactly at instant \(t\) so that its treatment can be carried out. A job can therefore be seen as an instance of a task.

1.2.2.1 The deadline of a real-time task

One of the key parameters of a real-time task \(\tau_i\) is naturally its relative deadline noted \(D_i\) which reflects the timing constraint on its execution. This quantity can be expressed as a number of CPU clock cycles but other reference units can be used, such as CPU ticks\(^2\) for instance. Hereafter, we will use the term “time unit” to refer to the used reference unit. In this thesis, \(D_i\) denotes the relative deadline of \(\tau_i\) (i.e., relative to its last job release), with the interpretation that once the task releases a job, that job must be completely executed by \(D_i\) time units.

1.2.2.2 The period of a real-time task

Another key parameter of any task \(\tau_i\) is its period denoted by \(T_i\). There exist three distinct interpretations of this parameter, each leading to a well-defined type of task. According to the interpretation given to the period, tasks can be classified into three categories:

\(^1\)An application is said to be “schedulable” if it can execute without missing any deadline.
\(^2\)A processor tick is a periodic interrupt generated by the OS (for details, see Section 1.3.3 about the interrupts).
1.2 The application layer

1. *Periodic task.* The period $T_i$ denotes the exact delay between two consecutive job releases of $\tau_i$. Along with this interpretation, it is often assumed that the release time of the very first job of the task is also known beforehand, thus implying that the exact release time of every job can be computed at system design-time.

2. *Sporadic task.* The period $T_i$ denotes the minimal delay between two consecutive job releases of $\tau_i$. That is, the exact release time of every job is not known before they are actually released at run-time.

3. *Aperiodic task.* The task does not have a period parameter. That is, system designers have no prior information about the time-instants at which jobs are released.

Very often, the theoretical results proposed in the literature apply only to tasks that provide a particular relation between their period and deadline. Therefore, authors characterize a task $\tau_i$ using a specific vocabulary; $\tau_i$ is said to be constrained-deadline if $D_i \leq T_i$ or implicit-deadline in the particular case where $D_i = T_i$. When the proposed result holds whatever the relation between period and deadline, $\tau_i$ is said to be arbitrary-deadline. Note that the following inclusion holds: an implicit-deadline task is a constrained-deadline tasks which is in turn an arbitrary-deadline task. Thanks to this inclusive relation between the task models, any property that holds for an arbitrary-deadline task also holds for a constrained- and implicit-deadline task.

1.2.2.3 The worst-case execution time of a real-time task

The source code of a task, like any other code, can contains conditional treatments. For example, let $\tau_x$ be the task for which the pseudo-code is given in Algorithm 1. This task periodically reads a data from a sensor and executes a procedure $f()$ only when the data has the specific value $v$. Otherwise, if the data is different from $v$ (or after having executed $f()$), the task sleeps until the next period. In this example, each execution of the while-loop correspond to a job of $\tau_x$. That is, we say that a job completes when it enters the function “sleepUntilNextPeriod()” and a new job is released whenever the execution returns from this function (assuming that the first job is released when the first execution of this code starts). Clearly, depending on the value of the data, the time needed to complete two distinct jobs of this task can be different, and this difference can be very large depending on the execution time of the function $f()$.

Since real-time systems are designed to achieve only a few specific functions on a
CHAPTER 1. INTRODUCTION TO EMBEDDED REAL-TIME SYSTEMS

Algorithm 1: Simple code with a conditional treatment.

```
begin
  job_index = 1;
  while True do
    data = readDataFromSensor();
    if (data = v) then call f();
    sleepUntilNextPeriod();
    job_index++;
  end while
end
```

specific hardware, most researches on the real-time issue assume that the Worst-Case Execution Time (WCET) of every task is known beforehand. The WCET of a task $\tau_i$ indicates the largest execution time of any job of $\tau_i$, assuming that its execution is not interrupted. This third parameter is noted $C_i$ for each task $\tau_i$ and is usually expressed in the same reference unit as deadlines and periods. In the example above, the WCET $C_x$ of the task $\tau_x$ obviously includes the WCET of the function $f()$. Notice that the value of $C_i$ depends not only on the code of $\tau_i$, but varies from hardware to hardware. This quantity can be obtained by using software tools, but it can also be derived from practical measurements. There exists an abundant literature about the WCET analysis that covers numerous task models and considers various hardware architectures, but this topic falls out of the scope of this thesis. The interested reader may however refer to the surveys presented in [57, 66] for further investigations in this research field.

1.2.2.4 The offset of a real-time task

Another very common parameter of a real-time task $\tau_i$ is its offset noted $O_i$. The offset is the time delay before the release time of the first job of the task. In other words, the offset $O_i$ corresponds to the release time of the first job $\tau_{i,1}$ of task $\tau_i$. When the whole application is modeled by a single set of tasks (other application models are discussed in the Section 1.2.3) with identical offsets, the application is said to be synchronous; without loss of generality, the offset of every task can be considered as 0 and can be ignored. Otherwise, if $\exists \tau_i, \tau_j$ with $i \neq j$ and $O_i \neq O_j$, the application is said to be asynchronous. Notice that the offset of a task is defined only if the task is periodic. Indeed, by definition of a sporadic/aperiodic task, the release times of the jobs (including the first one) are not known beforehand.
1.2 The application layer

1.2.2.5 Notations and example of representation

In the remainder of this thesis, we will draw graphics similar to the one of Figure 1.4 in order to depict how the tasks are scheduled upon the processor(s). This figure illustrates the schedule of a single periodic constrained-deadline task $\tau_i = \langle O_i, C_i, D_i, T_i \rangle$ on a single processor noted $\pi_1$. The parameters of $\tau_i$ are: $O_i = 2, C_i = 3, D_i = 4$ and $T_i = 6$. Each red box represents a job of $\tau_i$ and its length corresponds to its worst-case execution time. The deadlines and release times are represented by down and up arrows, respectively. According to the definitions above, $\tau_i$ releases a job noted $\tau_{i,j}$ (with $j = 1, \ldots, \infty$) and the $k^{th}$ job of $\tau_i$ is $\tau_{i,k}$ at each instant $a_{i,j} \overset{\text{def}}{=} O_i + (j - 1) \cdot T_i$, each such job has a WCET of $C_i$ and must complete by its absolute deadline noted $d_{i,j} \overset{\text{def}}{=} a_{i,j} + D_i$.

Figure 1.4: Schedule of $\tau_i = (2, 3, 4, 6)$, where $\tau_i = (\text{offset, WCET, deadline, period})$.

1.2.2.6 Additional notions, definitions and notations

In the remainder of this document, we shall use the following definitions. Note that a reference is always mentioned into brackets next to the index of each result (lemmas, corollaries, etc.) if the result in question has been published in the literature. Otherwise, we do not annotate the result.

**Definition 1.1 (Active, running and waiting job)**

At any time $t$ during the system execution, a job $\tau_{i,j}$ is said to be active iff (if and only if) $a_{i,j} \leq t$ and it is not completed yet.
Besides, an active job is said to be running at time \( t \) if it is executing on any processor at time \( t \). Otherwise, the active job is said to be waiting. The notations active\((\tau, t)\), run\((\tau, t)\) and wait\((\tau, t)\) denote the subsets of active, running and waiting jobs of the tasks of \( \tau \) at time \( t \), respectively. The following two relations hold:

\[
\begin{align*}
\text{run}(\tau, t) \cup \text{wait}(\tau, t) &= \text{active}(\tau, t) \\
\text{run}(\tau, t) \cap \text{wait}(\tau, t) &= \emptyset
\end{align*}
\]

Informally, any job \( \tau_{ij} \) becomes active at its release time \( a_{ij} \) and remains so until it has executed for an amount of time equal to its execution requirement or until its deadline has elapsed.

**Definition 1.2 (Task utilization)**

We define the utilization \( U_i \) of a task \( \tau_i \) as the ratio between its worst-case execution time and its period, i.e.,

\[
U_i \overset{\text{def}}{=} \frac{C_i}{T_i}
\]

Informally, the utilization \( U_i \) of \( \tau_i \) denotes the highest proportion of time during which a processor could be used to achieve all the execution requirements of \( \tau_i \). For example, the task \( \tau_i \) in Figure 1.4 has an utilization of \( \frac{3}{6} = 0.5 \) and we can see that, from the time-instant 2, 50\% of the time is used for the execution of \( \tau_i \)'s jobs.

**Definition 1.3 (Generalized utilization and maximal utilization)**

The generalized utilization \( U_{\text{sum}}(\tau) \) and the maximal utilization \( U_{\text{max}}(\tau) \) of a set of tasks \( \tau \) are defined as follows:

\[
U_{\text{sum}}(\tau) \overset{\text{def}}{=} \sum_{\tau_i \in \tau} U_i
\]

\[
\text{and } U_{\text{max}}(\tau) \overset{\text{def}}{=} \max_{\tau_i \in \tau}(U_i)
\]
1.2 The application layer

Definition 1.4 (Task density)

We define the density \( \delta_i \) of a task \( \tau_i \) as the ratio between its worst-case execution time and the minimum between its deadline and its period, i.e.,

\[
\delta_i \overset{\text{def}}{=} \frac{C_i}{\min\{D_i, T_i\}}
\]

Definition 1.5 (Generalized density and maximal density)

Similarly to \( U_{\text{sum}}(\tau) \) and \( U_{\text{max}}(\tau) \), we define the generalized density \( \delta_{\text{sum}}(\tau) \) and the maximal density \( \delta_{\text{max}}(\tau) \) of a set of tasks \( \tau \) as

\[
\delta_{\text{sum}}(\tau) \overset{\text{def}}{=} \sum_{\tau_i \in \tau} \delta_i \quad \text{and} \quad \delta_{\text{max}}(\tau) \overset{\text{def}}{=} \max_{\tau_i \in \tau} \{ \delta_i \}
\]

Definition 1.6 (Demand bound function [18])

The demand bound function \( DBF(\tau_i, t) \) of a sporadic task \( \tau_i \) provides an upper bound on the cumulative execution time of the jobs of \( \tau_i \) that are both released in, and have deadline within, any interval of length \( t \). This demand bound function is given by

\[
DBF(\tau_i, t) \overset{\text{def}}{=} \max \left\{ 0, \left( \frac{t - D_i}{T_i} \right) + 1 \right\} \cdot C_i
\]

Definition 1.7 (Application load [18])

Based on the demand bound function, the load parameter is defined for any set of tasks \( \tau \) as follows:

\[
\text{LOAD}(\tau) \overset{\text{def}}{=} \max_{t > 0} \left\{ \frac{\sum_{\tau_i \in \tau} DBF(\tau_i, t)}{t} \right\}
\]
CHAPTER 1. INTRODUCTION TO EMBEDDED REAL-TIME SYSTEMS

1.2.2.7 Simple example of a real-time application

The notations and definitions given above are the basis of almost every research in the real-time scheduling theory. Hence, before going any further let us consider a simple (fictive) example of real-time system in order to provide a good understanding of the task models and to familiarize the reader with these notations. The considered example is a sprinkler control system. Suppose that this system is installed in the ceilings of a 3-floors building as illustrated in Figure 1.5a. There are 9 fire sensors (each coupled up to a sprinkler) and one main controller. The main controller is installed in the basement and the sensors are installed all around the building (3 sensors at each floor) and each one periodically informs the main controller of the absence/presence of a fire. If any sensor comes to detect a fire (see Figure 1.5b), it sends the information to the main controller which turns on the corresponding sprinkler, as well as all the sprinklers located at the same floor (see Figure 1.5c). We assume that

- the system requests the sensors every second and it takes at most 50 milliseconds to transfer the information from each sensor to the main controller,
- the main controller takes at most 500 milliseconds to determine the neighborhood of the alarmed sensor (if a fire is detected) and to turn on all of them,
- all the functionalities of the application start simultaneously after the system has been booted up.

The whole application described above can be modeled by a single set of 10 periodic implicit-deadline tasks: 9 tasks $\tau_1, \tau_2, \ldots, \tau_9$ that request the sensors ($\forall i = 1, \ldots, 9$: $O_i = 0, C_i = 50$ and $D_i = T_i = 1000$) and one task $\tau_{10}$ that implements the main controller (with $O_{10} = 0, C_{10} = 500$ and $D_{10} = T_{10} = 1000$). That is, every task $\tau_i$ releases a job noted $\tau_{i,j}$ ($j = 1, \ldots, \infty$ and the $k$th job of $\tau_i$ is $\tau_{i,k}$) at each instant $a_{i,j} \overset{\text{def}}{=} (j - 1) \cdot T_i$, each such job has a WCET of $C_i$ and must complete by its absolute deadline noted $d_{i,j} \overset{\text{def}}{=} j \cdot T_i$.

Assuming that this application is running on a single-processor platform, Figure 1.6 illustrates a possible scenario of its execution: the 9 tasks requesting the sensors (depicted in blue) are executed in the order $\tau_1 \rightarrow \tau_2 \rightarrow \cdots \rightarrow \tau_9$, followed by the execution of the controller task (depicted in red). After 1 second (at time 1000 assuming that the millisecond is the reference unit), every task releases a new job and the schedule repeats itself continuously. By repeating this execution pattern, one can easily see that every job completes by its deadline.
1.2 The application layer

(a) 9 sprinklers, 3 at each floor, and a main controller installed in the basement.

(b) The left sprinkler at the second floor detects a fire.

(c) The main controller activates all the sprinklers of the second floor.

Figure 1.5: A sprinkler system installed in a 3-floor building.
CHAPTER 1. INTRODUCTION TO EMBEDDED REAL-TIME SYSTEMS

Figure 1.6: A possible execution scenario of the sprinkler system on a single-processor architecture.

Figure 1.7: Illustration of a multimode real-time application composed of 5 operating modes. The arrows that link the modes represent the mode transitions that can occur at run-time. Here, mode transitions are possible between every mode.

1.2.3 Multimode application model

Very often, a real-time application is denoted by $\tau$ and is modeled by a single and fixed set of $n$ real-time tasks, i.e., $\tau \overset{\text{def}}{=} \{\tau_1, \tau_2, \ldots, \tau_n\}$. This traditional model of application has been used in the example of the previous section. However in practice, applications are usually designed so that they are optimized to cope with all the requirements and all the situations generated by the environment. That is, practical applications are often designed as several operating modes, where each mode provides a dedicated behavior or serves a specific purpose, e.g., an initialization mode, an emergency mode, a fault
1.2 The application layer

recovery mode, etc. Such applications are referred to as multimode application in the remainder of this document.

In multimode applications, it is not rare that some modes contain “unique” functionalities—tasks that belong to only one mode and which are mutually exclusive and independent from the other tasks of the same or other modes—and modeling such applications in a conventional manner (i.e., by a single set of tasks) does not reflect this probable independence between the modes and their respective tasks. Therefore, a multimode real-time application \( \tau \) is modeled by a set of \( x \) operating modes noted \( M_1, M_2, \ldots, M_x \) where each mode contains its own set of functionalities to execute, i.e., its set of tasks. That is, a mode \( M_k \) contains a set \( \tau_k \) of \( n_k \) tasks denoted \( \{ \tau_{k1}, \tau_{k2}, \ldots, \tau_{nk} \} \).

Informally, a multimode application can be seen as a set of sets of real-time tasks. Figure 1.7 illustrates a (fictive) multimode application composed of 5 modes, each one having its own set of tasks to execute.

At any time during the execution of a multimode application, either the system runs in only one of its modes, i.e., it executes only the set of tasks associated with the current selected mode, or the system is switching from one mode to another one. A task must be enabled to generate jobs, and the system is said to run in mode \( M_k \) only if all the task of \( \tau_k \) are enabled and all the tasks of the other modes are disabled. In the following, we respectively denote by \( \text{enabled}(\tau_k, t) \) and \( \text{disabled}(\tau_k, t) \) the subsets of enabled and disabled tasks of \( \tau_k \) at time \( t \). Enabling a task \( \tau_{ki} \) allows it to generate jobs and disabling \( \tau_{ki} \) prevents future job releases from \( \tau_{ki} \). Thereby, when the system has to switch from the current mode to any other one, it basically disables all the tasks of the current mode and enables those of the destination mode. This substitution introduces a transient stage where the jobs of the current and destination modes may be active simultaneously. During such transition phases, the system could be overloaded and the system reliability can be compromised. This problem is thoroughly studied in Chapter 2 where we propose two different transition protocols to solve it. The next section provides the reader with an overview of the second conceptual layer of any computer system, i.e., the Operating System.
CHAPTER 1. INTRODUCTION TO EMBEDDED REAL-TIME SYSTEMS

1.3 Real-Time Operating Systems (RTOS)

1.3.1 Overview of the main functionalities

An operating system (OS) can be roughly defined as a software—programs and data—that provides an interface between the hardware and the other user processes. As mentioned in Section 1.1.3, the main role of an operating system is to act as a host for the application layer, to manage processes and to coordinate the allocations and sharing of the hardware resources, such as the processor(s) and memory for instance. In addition, the OS enables the user applications to manipulate the hardware via several software routines. This relieves the application programmers from having a deep knowledge of the hardware specification. These software routines provide many kinds of services and are gathered in Application Programming Interfaces (APIs). By invoking these routines, any user application can request a service from the operating system, pass parameters, and receive the results of the operation. Notice that in classical designs, the OS provides two distinct execution modes (unrelated to the application modes introduced in the previous section): a “protected” (or “kernel”) mode and a “user” mode, where processes running in kernel mode have priority over those running in user mode. Along with such design, an OS routine or service is typically executed in kernel mode whereas user processes are executed in user mode. The reason is that the OS routines must often manipulate hardware resources and a failure during the execution of such routines could cause more damage to the system than a failure brought about a user program; OS routines are considered as more critical than user processes.

A Real-Time Operating System (RTOS) is an OS intended for real-time applications, providing programmers with predictability and reliable control over process priorities. For now, we define the predictability as the ability to guarantee at system design-time that all the system requirements will be fulfilled at run-time. Note that this definition will be refined later, in Section 1.3.5. Since the tasks have to meet some timing requirements, minimal interrupt and process switching latencies are key factors of RTOS. For the same reason, the execution of any user process in an RTOS may have priority over the execution of an OS routine or service, contrarily to non-real-time OS. Typically, an RTOS is valued more for the accuracy of its predictability than for the amount of work that it can perform in a given interval of time. In other words, the most valuable quality for an RTOS is not to outperform other OS in terms of performance, but to provide accurate information about the time it takes to handle interruptions and to complete a job, so that
it increases the predictability and guarantees the respect of either soft or hard real-time constraints. A RTOS that can usually or generally meet a deadline is a soft RTOS, but if it can meet a deadline deterministically it is a hard RTOS.

Among the main responsibilities of an OS (including RTOS), one can cite the management of: processes, interrupts, physical and virtual memory, disk accesses, file systems, device drivers, networks and security. There are many things to say about each of these roles but we will only (and briefly) describe those that present an important difference between their real-time and non real-time implementation, namely, memory management, inter-task communication, resource management, interrupt management and processes management (also named scheduler).

1.3.2 Memory management

Memory allocation is much more critical in an RTOS than in other non real-time operating systems. As mentioned above, even though the speed of allocation is important, the most essential feature of RTOS is to ensure predictability. Consequently, a standard memory allocation scheme which scans a linked list of indeterminate length until finding a suitable free memory block is appropriate for non real-time OS but unacceptable for RTOS. Rather, RTOS have to allocate memory in a bounded time and the fixed-size-blocks algorithm works astonishingly well for simple embedded systems. A brief description of this algorithm is given below.

*Fixed-size-blocks allocation, or memory pool allocation,* uses a free list of fixed-size blocks of memory and allows dynamic memory allocation. A memory pool is defined as a set of preallocated memory blocks of equal size, each represented by a number called “handle” (see Figure 1.8). Based on such handles, the application can allocate, access, or free the corresponding memory blocks at runtime. A memory pool module is composed of several memory pools which differ from each another in the size of their memory blocks. For instance, a memory pool module can allocate 3 memory pools at compile time, with block sizes optimized for the application. The application can allocate, access and free memory blocks via an interface such as the (fictive) following one:

- **memAlloc(required_size)**. Allocates a memory block from the pools and returns the corresponding handle. This function determines the first pool in which blocks have a size larger than the specified one. If all blocks of that pool are already
reserved, the function tries to find another free block in the next bigger pool(s). An allocated memory block is represented with a handle. For instance, every handle can be implemented with an unsigned integer and the module can interpret the handle by dividing it into pool index, memory block index and a version. Based on such handle implementation, the pool and the memory block index allow fast access to the corresponding block, while the version, which is incremented at each new allocation, allows the detection of handles for which the corresponding memory block is already freed.

- `memGet(handle)`. Get an access pointer to the allocated memory.
- `memFree(handle)`. Free the allocated memory block pointed by the specified handle.

Strategies using memory pools benefit from the following advantages:

1. Since the number and size of the memory pools are known at compile-time, accessing, freeing or looking for a free block can be done in an upper-bounded execution time, hence maintaining the execution predictability.
2. Assuming that a flag is associated to each pool in order to indicate whether all its blocks are free, the memory release for thousands of objects in a pool can be done in just one operation (by flipping this flag), not one by one if operations such as “malloc” have been used to allocate memory dynamically for each object.

3. Memory pools can be grouped in hierarchical tree structures, which is suitable for special programming structures like loops and recursions.

4. Fixed-size block memory pools do not need to store allocation metadata for each allocation, describing characteristics like the size of the allocated block. Particularly for small allocations, this provides a substantial space savings.

Among the downsides of such strategies, one can cite the fact that memory pools often need to be tuned for the application which deploys them, hence requiring much more time for designing the application.

1.3.3 Interrupt handlers

An interruption can be defined as an asynchronous signal (usually generated by the hardware) that indicates a need for attention, but it can also be a synchronous event generated by the software in order to indicate a need for a change in the execution. In both cases, it causes the operating system to suspend its current activity in order to serve the Interruption Request (IRQ). Among the numerous types of interrupt, one can cite for instance: system timers, disks I/O and power-off signals. System timer interrupts are mainly used to keep time by interrupting the system periodically—providing the OS with a metric of time that we called “CPU ticks” in Section 1.2.2—but they can also be used to provide the RTOS with some points in time at which tasks must be rescheduled. A disk I/O interrupt signals the completion of a data transfer from/to the disk peripheral. Finally, a power-off interrupt predicts or requests a loss of power, allowing the system to react by an appropriate action, such as switching to a low-power mode or performing an orderly shutdown, typically. Other interrupts exist in order to transfer data bytes on a network, sense key-presses, control motors or anything else that the device must do.

In typical designs, whenever an IRQ occurs, the processor saves its state of execution and starts the execution of an interrupt handler. Since an interrupt handler blocks the highest priority task from running, and since RTOS are designed to keep system overheads to a minimum, interrupt handlers of RTOS are kept as short and fast as
possible. Hence, the interrupt handler of most RTOS defers all interaction with the hardware as long as possible; typically all that is necessary is to acknowledge or disable the interrupt (so that it will not occur again when the interrupt handler returns). The interrupt handler then queues the work to be done at a lower priority level, often by unblocking a driver task through releasing a semaphore or sending a message (see the next section for details about semaphores and messages). Notice that processors typically have an internal interrupt mask which enables the software to ignore all external hardware interrupts while being set.

1.3.4 Inter-task communication and resource sharing

A second main functionality of an OS is to handle inter-task communication and resource sharing. When the OS can manage more than one task—sometimes named “Multitasking OS”—it must manage sharing data and hardware resources among them. Nowadays, it is well-known that it is usually “unsafe” for two tasks (or more) to simultaneously access the same data or the same hardware resource (“unsafe” means that the results can be inconsistent or unpredictable, especially if a task is modifying a data collection, then the view of another task is correct either before the changes begin or after the changes are completely finished). Basically, the most popular shared resource protocols, but not every of them, ensure the three following properties.

a) **Exclusive access.** This property ensures that any two tasks can never access the same resource simultaneously.

b) **Starvation-free.** We say that a starvation occurs when a task indefinitely waits for a resource and never gets it. For example, this phenomenon occurs when a task holds a resource and never releases it (if the process has failed for instance). That is, another task waiting for this resource will never be allowed to use it, leading to a starvation.

c) **Deadlock-free.** A deadlock can be seen as a crossed starvation. In the simplest deadlock scenario, two tasks are holding two distinct resources and both are trying to access to the resource held by the other task. Formally, a task $\tau_A$ holds a resource $R_1$, another task $\tau_B$ holds another resource $R_2$ and $\tau_A$ and $\tau_B$ try to access to $R_2$ and $R_1$, respectively. Since each task is waiting for the resource held by the other one without releasing the resource that it holds, it will cause the tasks to wait indefinitely; the system is said to be frozen. Deadlocks are usually prevented by careful design or by using a suitable protocol.
1.3 Real-Time Operating Systems (RTOS)

In practice, three distinct paradigms are generally used to deal efficiently with shared resources. Some of them do not ensure the three properties cited above, because sometimes, scenarios such as deadlock never occur (actually, it depends on the tasks behavior and resources characteristics). These three paradigms are listed and briefly described below.

1. Temporarily masking/disabling interrupts.
2. Binary semaphores.

1. Temporarily masking/disabling interrupts. Usually, non real-time OS do not allow user programs to mask (disable) interrupts, because the user program could control the CPU for as long as it wishes. Furthermore, modern CPUs do not allow a code running in user-mode to disable interrupts neither, because they consider such control as a responsibility of the operating system. However, contrarily to non real-time OS, many RTOS allow the application itself to run in kernel mode (also known as “super-user mode”) for greater system call efficiency and also to enable the application to have greater control of the system environment without requiring any OS intervention. In practice, disabling interrupts appears to be the best solution (i.e., providing lowest overhead) to prevent from simultaneous access to a shared resource. However care must be taken not to dramatically increase the system interrupt latency. Indeed, in the case of a single-processor system, a task running in kernel mode and disabling the interrupts gets an exclusive use of the CPU since no other task or interrupt can take control. Therefore the critical section\(^1\) is protected but this section must obviously be shorter than the desired maximum interrupt latency, otherwise this method increases this maximum latency. Typically this technique is used only when the critical section is just a few instructions and does not contain any loop-code.

As mentioned above, masking/disabling interrupts is the most appropriate paradigm for RTOS, because it allows the application processes to have greater control of the system while reducing the overheads due to the OS interventions. However, when the critical section is longer than a few source code lines or involves lengthy looping, RTOS have to use mechanisms similar to those available on non real-time OS, such as semaphores and **OS-supervised inter-task messaging**. Such mechanisms involve system

\(^1\)The portion of code during which the task has an access to the shared resource.
calls, and usually invoke the operating system on exit, so they typically take hundreds of CPU instructions to execute while masking interrupts may take as few as one instruction on some processors. But for longer critical sections, there may be no choice since interrupts cannot be masked for long periods without increasing the system interrupt latency. These two other mechanisms are described below.

2. Binary semaphores. A binary semaphore is a binary variable associated to each shared resource. A semaphore is either locked or unlocked, with the interpretation that a task locks the corresponding semaphore before accessing a resource and unlocks it after its treatment on this resource. When a semaphore is locked, tasks that want to access the corresponding resource must wait for the semaphore. Typically a task can set a timeout on its wait for a semaphore. A well known problem with the semaphore paradigm is the priority inversion problem, i.e., a high priority task \( \tau_A \) is waiting because a low priority task \( \tau_B \) has locked a semaphore. To resolve this problem, there are several semaphore-based solutions, such as priority inheritance protocols for instance (see [62] for details about such approaches). A priority inheritance protocol temporarily inverses the priority of \( \tau_A \) and \( \tau_B \) so that \( \tau_B \) can execute until it releases the semaphore. Although it may sound easy to implement, handling multiple levels of inheritance can actually become tortuous and the main drawback of this paradigm is to become very complex when there are multiple levels of waiting, i.e., a task waits for a semaphore locked by another task, which waits for another semaphore locked by a third task and so on.

3. Message passing. The third most popular approach to deal with the problem of resource sharing is for tasks to send messages through an organized message passing interface (such MPI [35, 36] for instance). In this paradigm, the resource is managed by only one task. When another task wants to access to the resource, it requests the managing task by sending it a message. Although the real-time behavior is less crisp than semaphore-based protocols, simple message-based protocols avoid most deadlock hazards, and have generally a better behavior than semaphore-based protocols; in particular, a queue entry is the entire resource consumed by a waiting service request. However, problems like those of semaphores are possible: priority inversion can occur when a task is working on a low-priority message, and ignores a higher-priority message (or a message originating indirectly from a high priority task) in its incoming message queue. Deadlocks can also occur when two or more tasks are waiting for each other to send response messages.
1.3 Real-Time Operating Systems (RTOS)

1.3.5 Real-time schedulers

1.3.5.1 Role and history

In order to ensure that all the task deadlines and requirements are satisfied during the execution of the application, RTOS integrate a specific algorithm named the “scheduler” which schedules the tasks upon the processor(s). That is, the goal of every scheduler is to determine in which order tasks have to be executed upon the processor(s) so that all the job deadlines are met.

The real-time scheduling problem has been widely studied in the literature, where the first studies focused on periodic task sets. In 1969, Liu posed in [48] the scheduling problem of such task sets upon multiprocessor platforms, i.e., platforms composed of several processors available to the execution of the job generated by the periodic tasks (with the constraint that an individual job may execute on at most one processor at any instant in time). In this seminal paper, Liu identified a set of properties for periodic task sets which are sufficient (albeit not necessary) to guarantee feasibility upon an \( m \)-processor multiprocessor platform, i.e., any periodic task system satisfying these properties can always be scheduled upon an \( m \)-processor platform to meet all deadlines. Then, several other scheduling algorithms have been developed while considering the same task and platform model as Liu. Among the most seminal ones, one can cite the Slack-Time algorithm proposed by Leung in 1989 [46] or the task-to-cpu assignment strategies proposed by Burchard et al. in 1995 [24]; however, none of these algorithms were optimal in the sense of successfully scheduling all feasible periodic task systems.

In 1996, Baruah et al. presented in [14] necessary and sufficient feasibility conditions, together with an optimal scheduling algorithm based on the notion of PFAIR scheduling (Proportionate FAIRness). This algorithm—well known by the name of PF for Proportionate Fair—is the first global and optimal algorithm for scheduling real-time applications on multiprocessor platforms, where the tasks are assumed to be periodic and have implicit deadlines. The idea is to divide the time into time slots and to schedule every task in such a way that each one receives an amount of time slots proportional to its utilization. This was named the “fairness” property in [14]. Different adaptations of PF were published these past few years and among the most interesting ones, one can cite PD [15] and PD² [7]. These adaptations follow the same idea than PF but succeeded to considerably reduce the computing complexity of the scheduling decisions. The worst
drawback of all “Pfair-like” approaches resides in the fact that a scheduling decision must be taken at each time unit. That is, although the theoretical correctness of these algorithms PF, PD and PD\textsuperscript{2} is undeniable, their application in the real world is limited due to the high number of overheads (and especially the preemption overheads) that are generated during the execution of the application. Notice that, during an overhead on any processor (say \(\pi_k\)), only the Operating System is running while the tasks supposed to be executed on \(\pi_k\) are waiting for execution. Therefore, it exist some intervals of time in the actual schedule where a task is not executed while it was supposed to be executed in the theoretical schedule. Since most of the existing schedulability analyses do not take overheads into account, a task can sometimes miss its deadline even if the system was asserted to be schedulable in theory. This is depicted through a simple example in Figure 1.9. In that example, two tasks are executed on a single processor. At time 5, the job \(\tau_{x,y}\) is released and we assume that this job has a higher priority than \(\tau_{i,j}\). Therefore, \(\tau_{i,j}\) is preempted and resumed later when \(\tau_{x,y}\) completes. Figure 1.9a illustrates the schedule assumed in theory, in which OS does not bring about any overhead. On the other hand, Figure 1.9b illustrates the schedule which is actually produced at run-time. As we can see, although this application is asserted to be schedulable in theory, \(\tau_{i,j}\) misses its deadline in the produced schedule because of the overheads generated by the Operating System while preempting \(\tau_{i,j}\).

More recently (in 2003), Zhu, et al., addressed the problem of the preemption overheads and designed a new class of algorithms named “Boundary fair schedulers” (i.e., Bfair-like algorithms). Their idea is to ensure the fairness only at boundaries, i.e., some particular events occurring during the system execution, resulting in a considerable reduction of the number of decisions taken by the scheduler, as well as the cost of the overheads, compared to Pfair-like algorithms. Among the most interesting algorithms in this category, one can cite BF [69] (Boundary Fair), LLREF [28] (Largest Local Remaining Execution First), EKG [10] and more recently, an optimization of LLREF proposed by Shelby Funk and Vijaykant Nadadur in [33] and called LRE-TL. Basically, these “Bfair-like” schedulers can be divided into two categories: the “continuous-time” Bfair schedulers, i.e., the time is assumed to be continuous, and the “quantum-based” Bfair schedulers that divide the time into time slots. To the best of our knowledge, the most recent approach belonging to the continuous-time category is LRE-TL [33]. This algorithm was proved to be optimal but, like every scheduler that belongs to this category, it suffers from another drawback. Indeed, such schedulers can sometimes preempt a task \(\tau_i\) by another task \(\tau_j\) only to execute an extremely small portion of \(\tau_j\). That is, the
1.3 Real-Time Operating Systems (RTOS)

Figure 1.9: Because most of the existing schedulability analyses do not take overheads into account, a job can sometimes miss its deadline even if the system was asserted to be schedulable in theory.

generated schedule can undergo a preemption overhead only to execute an insignificant portion of a task, hence producing schedules in which the time actually used to execute the tasks is limited. On the other hand, quantum-based techniques have been designed in order to ensure that the Operating System only preempts a task $\tau_i$ by another task $\tau_j$ if $\tau_j$ has to execute at least one time-quantum. In our opinion, the most interesting study in this category is [69]. The proposed scheduler BF is optimal and it considerably reduces the number of preemptions compared to continuous-time Bfair and Pfair-like schedulers. However, the complexity of BF is $O(n \cdot T_{\text{max}})$ where $T_{\text{max}}$ is the largest period of the tasks$^{1}$, but since $T_{\text{max}}$ could be quite large, the overhead due to the execution of the scheduler can be large as well.

$^{1}$Although the authors claimed that the sub-routine responsible for this $T_{\text{max}}$ factor can be replaced by another routine running in $O(1)$ (leading to a complexity of $O(n)$ for BF), the proof of the equivalence between these two sub-routines does not appear in the literature and, in our opinion, this equivalence is not obvious at all.
Finally, Greg Levin et al. presented this year (2010) a simple model for understanding optimal multiprocessor scheduling [47]. They introduced the notion of DP-FAIR (Deadline Proportionate fair) and proposed the algorithm DP-WRAP. Additionally, they proved that DP-WRAP is optimal for periodic tasks and multiprocessor platforms composed of identical processors, and they demonstrated the flexibility of the DP-FAIR guidelines by extending DP-WRAP to handle sporadic task sets with arbitrary deadlines.

1.3.5.2 Classification of real-time schedulers

Due to the wide variety of system constraints and requirements, there exists various scheduling paradigms. Many papers propose a classification of the scheduling problems and algorithms (see [25, 34] for instance) but in our opinion, all these classifications are appropriate only until new algorithms are proposed. In this thesis, we decided to classify the scheduling algorithms into two broad categories: priority-driven schedulers and time-driven schedulers.

**Priority-Driven schedulers.** We define a priority-driven scheduler as a scheduler that bases its decisions on the notion of jobs priority. These scheduling algorithms are typically implemented as follows: at each instant in time, a priority is assigned to each active job and the scheduler allocates the available processor(s) to the highest-priority job(s). Note that this definition is not the same as the one proposed in [34, 39]. Different scheduling algorithms differ from one another in the manner in which priorities are assigned to individual jobs. Following our interpretation, algorithms such as Rate-Monotonic [49] (RM), Earliest Deadline First [30, 49] (EDF) or Least Laxity First [52] (LLF) are priority-driven algorithm.

**Time-Driven schedulers.** Such schedulers are invoked when a system timer comes to expire and have the responsibility to reset it, thereby deciding on the next scheduler invoking time. Typically, such schedulers are not priority-based. At each timer expiration, the scheduler reads an entry in a given table that we call the “schedule-table”. This table provides the scheduler with both the next job(s) to execute on the processor(s) and the next instant at which the scheduler must be re-invoked. In short, the schedule is generated according to this schedule-table. This scheduling paradigm is often used in the practical world when all the characteristics of the application are known beforehand (in particular, for periodic and implicit-deadline tasks). If the schedule-table is
1.3 Real-Time Operating Systems (RTOS)

completely determined at system design-time then one of the main advantages of such schedulers is typically to provide a low time-complexity at run-time (they generate low time-overheads during the execution of the application).

One of the fundamental differences between these two scheduling paradigms resides in their “flexibility” in the sense that, if at run-time a job completes in less time than its WCET (this frequently happens in practice) then priority-driven schedulers will (typically) directly start (or resume) the execution of the next waiting job with the highest priority. On the contrary, time-driven schedulers have to wait until the next invoking time before executing the next job. Regardless of its category (i.e., priority- or time-driven), real-time paradigms can also be broken into two broad categories: offline schedulers and online schedulers.

In an online scheme, the scheduler has to perform computations at run-time. By computation, we mean:

- compute the job priorities; this obviously concerns priority-driven schedulers including (for instance) EDF, LLF [52], PF [14], PD [15], PD\(^2\) [7], LL-REF [28] or LRE-TL [33].

- generate the schedule-table; typically, online time-driven schedulers determine only a local (partial) schedule-table that provides the schedule between the current time and the next scheduler invoking time. Some examples of such strategies include BF [69] or more recently DP-WRAP [47].

In an offline scheme, these computations are performed at system design-time. For priority-driven schedulers, this means that the job priorities are computed at system design time and are assigned to the tasks (or to the jobs) before the execution of the application. Usually, priorities are hard-coded in the task (or job) descriptors of the RTOS and the scheduler ensures at run-time that a low priority job is never running while a high priority job is waiting for execution. Typically, if the priorities are assigned to the tasks then at run-time every job inherits from the priority of the task it is issued from. Examples of such offline priority-based schedulers include the Rate Monotonic [49]. Concerning time-driven schedulers, offline schedulers determine the whole schedule-table at system design-time. Since this can be achieved only if all the tasks characteristics are known beforehand, such strategies are mainly used for periodic tasks. The main advantage of such strategies is to benefit from a very low computing
CHAPTER 1. INTRODUCTION TO EMBEDDED REAL-TIME SYSTEMS

complexity at run-time since the only role of the scheduler is to read the schedule-table and dispatch the specified jobs to the specified processors.

In most papers, priority-driven schedulers are classified according to the manner in which they assign priorities and three classes of priority assignment are usually distinguished:

1. Fixed Task-level Priority (FTP) assignment. A priority is assigned to each task and remains constant during the whole execution of the system. Such priority assignments are offline schedulers and a typical example of such schedulers is Rate Monotonic [49] (RM).

2. Fixed Job-level Priority (FJP) assignment. In this scheme, a priority is assigned to each job upon its release and remains constant until the job completes. Both offline and online schedulers can support this type of priority assignment. Online schedulers compute the priority of every job upon its release (and sometime requires an offline initialization phase) whereas offline schedulers compute and store the priority of every job within a hyper-period\(^1\) at system design time (this obviously concerns periodic tasks). A typical example of online FJP scheduler is EDF [30, 49].

3. Dynamic Job-level Priority (DJP) assignment. The priority of any job can change at any time during the system execution. Notice that decreasing the priority of a running job may cause the scheduler to have to substitute the running job with another one with a higher priority. Some examples of such online strategies include LLF [52], LL-REF [28] or LRE-TL [33].

As we can notice, it holds from the interpretations given above that FTP schedulers are particular cases of FJP schedulers and FJP schedulers are particular cases of DJP schedulers. From an implementation perspective, there are significant advantages in using priority-driven FJP algorithms (thus including FTP algorithms); while it is beyond the scope of this thesis to describe in detail all these advantages, some important ones (cited in [34]) are listed below.

- Very efficient implementations of priority-driven FJP scheduling algorithms have been designed (see [53] for example).

\(^1\)The hyper-period of a real-time application is the least common multiple of the periods of its tasks.
1.3 Real-Time Operating Systems (RTOS)

- It can be shown that when a set of jobs is scheduled using a priority-driven FJP algorithm then the number of preemptions occurring during the execution of any job is bounded from above by the number of jobs in the set (and consequently, the total number of context switches is bounded by twice the number of jobs).

- It can similarly be shown that the total number of interprocessor migrations of individual jobs is bounded from above by the number of jobs.

From all these advantages, and because FJP schedulers have been widely studied in the literature, the energy-aware methods that we present in this thesis focus especially on such priority-driven FJP (and thus FTP as well) schedulers. The next section introduces some basic definitions about the scheduling algorithms.

1.3.5.3 Some basic definitions

Regardless of its category (i.e., priority- or time-driven, offline or online), real-time schedulers can provide (or not) the different features described below.

**Definition 1.8 (preemptive scheduler)**

The execution of any job may be suspended and resumed later if the scheduler has to execute a higher priority job meanwhile.

Targeting the problem of scheduling computer programs on processor(s), schedulers can be quite rightly considered as preemptive since most RTOS support multitasking (i.e., they are able to run more than one task) and processors generally enable to swap the current execution context for another one. However, in studies such as those concerning the scheduling of packets on a communication line, it can be reasonable to consider non-preemptive schedulers since sending a packet on a network is typically a non-preemptive operation.

**Definition 1.9 (An idle processor)**

A processor is idling if it is not running a task of the application.

In Figure 1.9a on page 31, the processor $\pi_1$ clearly idles during the time interval [15, 17]. In such a theoretical schedule, we will sometimes say in the remainder of this
thesis that the processor idles during each infinitesimal interval of time between the
instant at which the processor completes a job (or at time 0) and the instant at which the
scheduler dispatches a waiting job to that processor. That is, in Figure 1.9a $\pi_1$ idles at
instants 0, 5 and 10.

**Definition 1.10 (work-conserving scheduler [29])**

A work-conserving scheduler is a scheduling algorithm that does not let a proces-
sor idling while there is at least one job awaiting execution, i.e., one waiting job.

Generally, preemptive schedulers are work-conserving. Indeed, in a problem such as
scheduling computer programs on processor(s) (in which preemptions are allowed),
idling a processor while a job is awaiting execution obviously results in delaying the
completion time of that job, as well as the completion time of the other waiting jobs.
On the contrary non-preemptive schedulers are usually non-work-conserving, because if
preemptions are not allowed (such as in the problem of scheduling packets on a network
for instance) then it can be useful to delay the sending of a lengthy packet in order to
wait for a short packet to be ready to send.

In the real-time issue, the tricky part is usually not to design scheduling rules, but
to formally prove that a set of scheduling rules will always allow the system to meet
all the timing requirements at run-time. This concern is named “scheduling analysis”
and it enables the system designer to predict beforehand whether the application can be
successfully scheduled by a given scheduling algorithm. Typically, scheduling analyses
provide both feasibility and schedulability tests, according to the definitions cited below.

**Definition 1.11 (Feasible schedule)**

A schedule is said to be feasible if and only if it meets all the timing requirements,
i.e., all the deadlines are met.

**Definition 1.12 (Feasibility test)**

Given the characteristics of a real-time application $\tau$ and a hardware architecture
$\pi$, a feasibility test is a mathematical condition that indicates a priori whether a
feasible schedule exists for $\tau$ on $\pi$. 36
1.3 Real-Time Operating Systems (RTOS)

**Definition 1.13 (Schedulability test)**

Given the characteristics of a real-time application $\tau$, a scheduling algorithm $S$ and a hardware architecture $\pi$, a schedulability test is a mathematical condition that indicates a priori whether the schedule provided for $\tau$ by $S$ on $\pi$ is feasible.

**Definition 1.14 (Job dispatching)**

We say that a job $\tau_{i,j}$ is dispatched at time $t$ if it passes from the waiting state to the running state at time $t$.

1.3.5.4 Multiprocessor schedulers

Nowadays, it is well-known that real-time multiprocessor scheduling problems are typically not solved by applying straightforward extensions of techniques used for solving similar uniprocessor problems. One of the most typical examples to illustrate the difference between multi- and uniprocessor scheduling problems is the presence of “scheduling anomalies” in multiprocessor schedules. In short, for a given priority assignment, we say that an anomaly occurs when a positive change in the system—a modification that should not jeopardize the feasibility such as adding processors, reducing a WCET, post-dating a job release or a task deadline, etc.—turns a feasible schedule into unfeasible. Deeply discussing the anomaly phenomenon goes out of the scope of this thesis and we refer the reader to [27] for more details. The reader should only keep in mind in the following that in multiprocessor environments, even an a priori insignificant modification in the task set leads the system designer to perform the schedulability analysis anew.

There are two main paradigms to solve the multiprocessor scheduling problem: *global* scheduling and *partitioned* scheduling. A global scheduling algorithm does not associate individual tasks to specific processors; rather the inter-processor migration of individual jobs is permitted (i.e., a preempted job may resume its execution on a different processor). On the contrary to global approaches, partitioned scheduling algorithms partition the set of tasks among the processors and schedule each partition on a single processor by an uniprocessor scheduling algorithm, i.e., partitioned approaches require that all the jobs generated by a task execute upon its assigned processor. In other words, global scheduling algorithms, on the contrary to partitioned algorithms, allow different tasks and different jobs to be executed upon different processors.
In typical designs, a task has three states: running, waiting and blocked. An executing task is “running” (only one task per CPU is running). Otherwise, the task is either blocked, i.e., awaiting a resource (other than the processor) held by another task, or waiting, i.e., waiting for the completion of a higher-priority running task. The OS stores the waiting tasks in a waiting list and the blocked tasks in a blocked list. That is, global schedulers have a single waiting list and a single blocked list for the whole multiprocessor platform (as illustrated in Figure 1.10a) whereas partitioned schedulers have one waiting and blocked list associated to each CPU (see Figure 1.10b).

There are advantages and disadvantages to both partitioned and global scheduling—for a discussion on a trade-off, please consult [8, 9]. Actually, it has been shown that the two paradigms are incomparable in the sense that, for a given priority assignment, there exists some set of tasks that are feasible using a global schedule and unfeasible with a partitioned schedule; inversely, there exists some set of tasks that are feasible with a partitioned schedule and unfeasible using a global schedule. To illustrate that, consider
1.3 Real-Time Operating Systems (RTOS)

the following fixed task-level priority (FTP) assignments and systems.

Figure 1.11: Schedule of $\tau_1 = \langle 2, 3 \rangle$, $\tau_2 = \langle 4, 6 \rangle$ and $\tau_3 = \langle 6, 12 \rangle$ (where $\tau_i = \langle C_i, D_i = T_i \rangle$) on a 2-processors platform with the global FTP assignment: $\tau_1 > \tau_2 > \tau_3$.

- Let $\tau = \{\tau_1, \tau_2, \tau_3\}$ be an application composed of 3 periodic implicit-deadline tasks with parameters $(\tau_i = \langle C_i, D_i = T_i \rangle)$: $\tau_1 = \langle 2, 3 \rangle$, $\tau_2 = \langle 4, 6 \rangle$ and $\tau_3 = \langle 6, 12 \rangle$, yielding $U_1 = U_2 = \frac{2}{3}$ and $U_3 = \frac{1}{3}$. Figure 1.11 shows that $\tau$ can be globally scheduled on a 2-processors platform by the FTP priority assignment $\tau_1 > \tau_2 > \tau_3$, where $\tau_i > \tau_j$ means that $\tau_i$ has a higher priority than $\tau_j$. However, there does not exist a partition of $\tau$ into 2 subsets such that the generalized utilization of each subset is not larger than one. Indeed, the three possible partitions are $\{\tau_1, \tau_2\}$, $\{\tau_3\}$, $\{\tau_1, \tau_3\}$ and $\{\tau_2, \tau_3\}$ and none of the subsets $\{\tau_1, \tau_2\}$, $\{\tau_1, \tau_3\}$ and $\{\tau_2, \tau_3\}$ can be scheduled on a single processor since their generalized utilization is larger than 1.

- On the other hand, consider the system $\tau = \{\tau_1, \tau_2, \tau_3\}$ with parameters: $\tau_1 = \langle 1, 2 \rangle$, $\tau_2 = \langle 2, 4 \rangle$, $\tau_3 = \langle 2, 3 \rangle$ and $\tau_4 = \langle 2, 6 \rangle$. That is, $\tau$ can be scheduled using the partitions $\{\tau_1, \tau_2\}$, $\{\tau_3, \tau_4\}$ but we can show that no global FTP assignment leads to feasible global schedule.

Among the most popular global, preemptive, work-conserving and FJP scheduling algorithms, one can cite Global Deadline Monotonic [11] and Global Earliest Deadline First (EDF) [11]. For sake of simplicity, Global Deadline Monotonic and Global Earliest Deadline First will henceforth be referred as global-DM and global-EDF in the remainder of this document. Global-DM is a FTP algorithm that assigns a static (i.e. constant) priority to each task according to their (relative) deadlines: the smaller the task deadline, the greater its priority. global-EDF is a FJP scheduler that assigns priority to jobs according to their (absolute) deadlines: the earlier the job deadline, the greater its priority.
1.4 The physical layer

1.4.1 Overview of the main components

The hardware layer contains the physical components of the system. It can include processors, memory, networks, antennas and/or any other kind of hardware devices. All these components are interconnected by buses, organized differently from system to system. Although we will only focus on the processors in the remainder of this document, memory and networks are nonetheless the subjects of thorough researches in the context of embedded real-time systems. More precisely, these researches focus on guarantying the strict respect of temporal constraints while data are read from/written to cache memory or while they are transmitted on a network (see [31] for instance). Note that, since embedded systems are designed for specific purposes, their components have to be adapted to their environment and are often subject to specific constraints. For example, memory in satellite must be tolerant to radiation [31], which implies high costs. Therefore, the size of memory that are embedded in satellites is limited, leading to the need of specific transmission protocols [31].

1.4.2 The different steps for manufacturing an integrated circuit

The goal of this section is certainly not to explain the full process of designing and building an Integrated Circuit (IC). Rather, our intention is to provide the reader with a basic knowledge and a succinct overview of this process. Roughly speaking, the process of designing IC can be divided into three successive steps: the logic design, the logical synthesis and the physical implementation.

The logic design is the initial step of the manufacturing process. In this process, the user creates the functional specification, using a variety of tools and languages including C/C++, SystemC, Simulink or MATLAB for instance. Basically, the user specification contains three types of information enumerated below.

1. It specifies the function of the chip, i.e., what the user wants the chip to do.
2. It specifies some constraints such as the maximum size of the chip, the desired performance, etc.
3. It contains information about the desired hardware such as (for instance): the number of CPUs, the type of the CPUs (RISC or CISC), the number of stages of
the pipelines, the technology node (65nm, 45nm, 35nm ...), the number of ALU (Arithmetic Logic Unit), etc.

The **logical synthesis** is an iterative process by which the user specification is ultimately converted into a Register Transfer Level (RTL) description. An RTL description is a level of abstraction in which the behavior of the circuit is defined in terms of the flow of signals (or transfer of data) between hardware registers, and the logical operations performed on those signals. In short, RTL abstractions are written in Hardware Description Languages (HDL) such as Verilog or VHDL to create high-level representations of a circuit, from which lower-level representations and ultimately actual wiring can be derived. The produced RTL describes the exact behavior of the digital circuits on the chip, as well as the interconnections to inputs and outputs. The logical synthesis is performed in two successive (and iterative) steps. First, an RTL description is produced from the user specification. Second, this RTL description is simulated (via software tools) in order to verify whether the user requirements are fulfilled. If so, the produced RTL is passed to the next step (the physical implementation). Otherwise, the RTL is modified accordingly and the simulation is redone. The simulation process converts the RTL description into a design implementation in terms of logic gates. This conversion into an optimized gate-level representation is carried out by taking two inputs: the RTL description and a *standard-cell library* specified by the system designer. A standard-cell library can be seen as a set of files that describe the technology to use, in particular the implementation of the basic logic gates (cells) like AND, OR, etc. Basically, any logic gate can be implemented in different ways; some implementations allow high performance but involve high power dissipation while other implementations allow low power dissipation but imply low performance. That is, the standard-cell library passed to the simulation process specifies the exact implementation of the fundamental logic gates (and sometimes contains the implementation of macro cells such as adders, muxes and flip flops). Based on the RTL description and the specified standard-cell library, the simulation process produces a gate-level representation from which the user requirements are roughly verified. Indeed, at this stage, the produced gate-level representation does not reflect the final circuit. Rather, it only allows to approximately estimate some physical properties of the circuit. For example, this gate-level representation provides the final number of logic gates and, although this information allows the designer to approximate the size of the final die, the exact size cannot be determined without the knowledge of the exact position of the gates over the die. Therefore, this approximated size can be very optimistic. The whole logical synthesis process is repeated as long as the constraints imposed by the user are not met and the final output of this process is...
CHAPTER 1. INTRODUCTION TO EMBEDDED REAL-TIME SYSTEMS

the RTL description that fulfills all these user requirements.

The physical implementation takes the RTL produced by the logical synthesis and creates the design of the chip. This process involves many steps, including defining places for the logic gates or wiring them together for instance. The physical implementation is a time consuming process during which the progression is not straightforward in practice. Numerous iterations are often required to ensure that all the objectives are met simultaneously. Here are some of the main steps of the physical implementation process:

- **Placement.** The logic gates are assigned to non-overlapping locations on the die area. Placement is an essential step in the process of designing the chip. An inferior placement assignment will not only affect the performance of the chip, but might also make it nonmanufacturable by producing excessive wirelength, which is beyond available routing resources. Consequently, a placer must perform the assignment while optimizing a number of objectives to ensure that a circuit meets its performance demands. Typical placement objectives include

  - Total wirelength: Minimizing the total wirelength, or the sum of the length of all the wires in the design, is the primary objective of most existing placers. This not only helps minimize chip size (and hence cost) but also minimizes power and delay, which are proportional to the wirelength.

  - Timing: The clock cycle of a chip is determined by the delay of its longest path, usually referred to as the critical path. Given a performance specification, a placer must ensure that no path exists with delay exceeding the maximum specified delay.

  - Congestion: While it is necessary to minimize the total wirelength to meet the total routing resources, it is also necessary to meet the routing resources within various local regions of the chip’s core area. A congested region might lead to excessive routing detours, or make it impossible to complete all routes.

  - Power: Power minimization typically involves distributing the locations of cell components so as to reduce the overall power consumption, alleviate hot spots, and smooth temperature gradients.

- **Clock insertion.** Clock signal wiring is (commonly, clock trees) introduced into the design. In a synchronous digital system, the clock signal is used to define a time reference for the movement of data within that chip. The clock distribution
network (or clock tree, when this network forms a tree) distributes the clock signal(s) from a common point to all the elements that need it. Since this function is vital to the operation of a synchronous system, much attention has been given to the characteristics of these clock signals and the electrical networks used in their distribution. Clock signals are often regarded as simple control signals; however, these signals have some very special characteristics. Among these special features, one can mention the fact that clock signals typically operate at the highest speeds of any other signal (including signal related to both control and data), within the entire synchronous system. Also, the clock waveforms must be particularly clean and sharp since the data signals are provided with a temporal reference by the clock signals. Finally, the control of any differences and uncertainty in the arrival times of the clock signals can severely limit the maximum performance of the entire system and create catastrophic conditions in which an incorrect data signal may latch within a register. The clock distribution network often takes a significant fraction of the power dissipated by a chip. Furthermore, significant power can be wasted in transitions within blocks, even when their output is not needed. These observations have led to a power saving technique called clock gating, which involves adding logic gates to the clock distribution tree, so portions of the tree can be turned off when not needed. The exact savings are very design dependent, but around 20-30% is often achievable.

- **Routing.** The wires that connect the gates are added. That is, the routing step adds wires needed to properly connect the placed components while obeying all design rules for the IC.

- **Postwiring optimization.** Performance, noise, and yield violations are removed via processes called **timing closure**, **signal integrity** and **design for manufacturability**, respectively. Timing closure is the process by which an IC design is modified to meet its timing requirements. Signal integrity is a measure of the quality of an electrical signal. In digital electronics, a stream of binary values is represented by a voltage (or current) waveform. Over short distances and at low bit rates, a simple conductor can transmit this with sufficient fidelity. However, at high bit rates and over longer distances, various effects can degrade the electrical signal to the point where errors occur, and the system or device fails. Signal integrity engineering is the task of analyzing and mitigating these impairments. Finally, design for manufacturability is the process by which the design is modified (where possible) to make it as easy and efficient as possible to produce. This process modifies the
design in order to make it more “manufacturable”, i.e., to improve its functional yield, parametric yield, or its reliability.

- **Final checking.** Since errors are expensive, time consuming and hard to spot, extensive error checking is the rule: (1) making sure that the mapping to logic was done correctly and (2) checking that the manufacturing rules were followed faithfully.

- **Tapeout and mask generation.** The design data is turned into photomasks. This is the final result of the design cycle for integrated circuits, the point at which the artwork for the photomask of a circuit is sent for manufacture. Note that the first tapeout is rarely the end of work for the design team. Most chips will go through a set of spins where fixes are implemented after testing the first article. Many different factors can cause a spin:

  - The taped out design fails final checks at the foundry due to problems manufacturing the design itself.
  - The design is successfully fabricated, but the first article fails functionality tests.

Depending on the failure (if any), the system designer has to restart the physical implementation from the corresponding step. In the worst-case scenario, the manufacturing process has to restart from the logical synthesis by modifying the RTL description. In the latter case, every subsequent step must be entirely redone.

### 1.4.3 Processors in embedded systems

Out of the *nine billion* processors manufactured in 2005, less than 2% were used by computers such as PCs, Macs and Unix workstations, while the other *8.8 billion* processors were incorporated into embedded systems [12]. Nowadays, processors are the core of every modern electronic device, spanning a large range of applications. “*From toys to traffic lights to nuclear power plant controllers, processors help run factories, manage weapon systems, and enable the worldwide flow of information, products, and people*” [12].

Embedded processors span the range from simple 4-bit microcontrollers, such as those in greeting cards or children’s toys, to powerful 128-bit microprocessors, such as specialized DSPs and network processors. Some of the products that include these chips run a short assembly program from ROM with no operating system, but many more
run real-time operating systems and complex C or C++ programs. It is also increasingly common to find variants of desktop-lite operating systems based on Linux and Windows controlling more powerful devices that are still clearly embedded systems [12].

Embedded processors can be broken into two broad categories: ordinary microprocessors and microcontrollers, which have many more peripherals on chip, reducing cost and size. Contrasting to the personal computer and server markets, a fairly large number of basic processor architectures are used; there are Von Neumann as well as various degrees of Harvard architectures, RISC as well as CISC and VLIW; word lengths vary from 4-bit to 64-bits and beyond (mainly in DSP processors) although the most typical remain 8/16-bit. Most architectures come in a large number of different variants and shapes, many of which are also manufactured by several different companies [12].

### 1.4.4 Power dissipation of a processor

#### 1.4.4.1 The usual model

In many former researches such as [37, 50, 58, 68] for instance, the power dissipation of a processor at any time \( t \) is estimated by the expression:

\[
Pwr(t) = \alpha \cdot f(t) \cdot V(t)^2
\]  

(1.1)

where \( \alpha \) is a parameter that depends on the considered processor and \( f(t) \) and \( V(t) \) are respectively the operating frequency and supply voltage of the CPU at time \( t \). According to Expression 1.1, reducing the frequency without reducing the voltage is useless. Indeed, if a code \( X_1 \) completes by at most \( C_1 \) time units at frequency \( f_1 \), then it completes by at most \( k \cdot C_1 \) time units at frequency \( \frac{f_1}{k} \). At frequency \( f_1 \), the energy consumption resulting from Expression 1.1 is \( E_1 = \int_{t}^{t+C_1} P(t) \cdot dt = C_1 \cdot \alpha \cdot f_1 \cdot V(t)^2 \) while at frequency \( \frac{f_1}{k} \), the energy consumption is \( E_2 = \int_{t}^{t+kC_1} P(t) \cdot dt = k \cdot C_1 \cdot \alpha \cdot \frac{f_1}{k} \cdot V(t)^2 = E_1 \). Thus, there is no energy gain.

In order to alleviate the power dissipation, the supply voltage \( V(t) \) must be reduced.

---

1Some papers assume that the power dissipation is simply a convex function of the processor speed
along with the frequency, but voltage and frequency are connected by the relation [55]

\[
\frac{1}{f(t)} \approx \frac{V(t)}{(V(t) - V_{th})^{\gamma}}
\]

where \(V_{th}\) is the gate-source threshold voltage, i.e., the voltage below which the current through the device drops exponentially. During the past decades, the threshold voltage of the manufactured devices was too high to generate a significant leakage current when the state of the device is off (see Section 1.4.4.3 for details), but still low enough compared to \(V(t)\) to be ignored in the above expression. That is, this expression above could be rewritten as

\[
f(t) \approx V(t)^{\gamma - 1}
\]

where the exponent \(\gamma\) is a constant depending on the considered technology. For instance, in technology based on classical MOSFETs (Metal-Oxide Semiconductor Field Effect Transistor), \(\gamma \approx 2\) [55], making the frequency a linear function of the supply voltage and the power \(P(t)\) a cubic function of the frequency. However, the exact knowledge of \(\gamma\) is not essential. The most important feature is the fact that the power function \(P(t)\) is a strictly increasing convex function of the frequency. That is, the results obtained by the methods presented in this thesis hold for any such functions, and the exact knowledge of the parameters of this function is therefore not essential. From Expression 1.2, reducing the voltage requires to reduce the frequency, and vice versa, but in practice several voltage levels can be used with a same frequency. In the remainder of this thesis, we thus assume that, when a frequency is selected, the voltage of the processor is reduced to the minimum supported voltage.

Although Expression 1.1 is used in many papers in order to model the power dissipation of the processor(s), its origin is hard to understand since it is mainly described in highly technical articles, which target an engineer public only. In the next chapters of this thesis, we will present various approaches that reduce different components of the processors consumption: the approaches presented in chapter 3 focus only on the static consumption (details are given in the next sections), whereas those presented in Chapter 4 focus only on the dynamic consumption. Because the consumption of a processor has multiple components, it is essential to provide the reader with a good understanding of (i) how processors consume energy and (ii) what are the key factors that influence their consumption. With this intention, the next sections explain the origin
of Expression 1.1 in familiar words. The interested reader may consult [38] for a more precise analysis of the consumption.

1.4.4.2 Power dissipation of the CMOS technology

Basically, a processor can be seen as a set of logic gates that carry out the fundamental logic functions, such as NOT, AND, or OR functions for instance. Consequently, the power dissipation of a processor can be approximated by the sum of the power dissipation of all its logic gates. Currently, CMOS (Complementary Metal-Oxide-Semiconductor) is the most popular technology for constructing integrated circuits such as logic gates [55]. CMOS technology is used in microprocessors, but also in microcontrollers, static RAM, and other digital logic circuits. Furthermore, CMOS technology is also used for a wide variety of analog circuits such as image sensors, data converters, and highly integrated transceivers for many types of communication.

CMOS-based devices have two important characteristics: their high noise immunity and their low static power dissipation. The static component of the power dissipation is explained in details below but in short, CMOS-based devices register a significant power dissipation only while the transistors in the device are switching between ON and OFF states. As a result, such devices do not produce as much waste heat as other forms of logic, such as TTL (Transistor-Transistor Logic) or NMOS logic (N-type metal-oxide-semiconductor). CMOS technology also allows a high density of logic functions on a chip. This was the main reason why CMOS technology became the most used technology to be implemented in VLSI chips; VLSI (Very-Large-Scale Integration) is the process of creating integrated circuits by combining thousands of transistor-based circuits into a single chip.

CMOS logic uses a combination of p-type and n-type metal-oxide-semiconductor field-effect transistors (MOSFETs) to implement logic gates and other digital circuits incorporated into computers, such as telecommunications or signal processing equipments. Commercial CMOS products are typically integrated circuits composed of millions (or billions) of transistors of both types on a rectangular piece of silicon of between 0.1 and 4 square centimeters. These devices are commonly called “chips” or “die”.

The main principle behind CMOS logic is to jointly use p-type and n-type MOSFETs
in order to create paths from either the voltage source or ground to the output. Depending on the input voltage of the component, some of its transistors are in a *blocking* state while others are *non-blocking*. If these blocking and non-blocking transistors create a path from the *voltage source* to the output, then the circuit is said to be *pulled up*. Otherwise, a path from the *ground* to the output is created and the output is therefore pulled down to the ground potential. Capital “C” of CMOS is due to this duality (Complementarity) between the blocking and non-blocking states of the transistors.

As mentioned above, since a processor can be seen as a set of numerous CMOS logic gates, the power dissipation of a processor can thereby be explained by examining the power dissipation of a single logic gate. With this intention, the next section considers the case of the simplest logic function: the NOT function. The power dissipation of other logic gates (such as AND, OR, NOR, etc.) is quite similar to this simple case study.

### 1.4.4.3 A case study: a standard NOT logic gate

A NOT logic function is often referred to as “input-inverter” circuit since the output signal of this circuit is the inverse of the input. That is, an input-inverter generates a low output voltage when the input voltage is high and reciprocally, it provides a high output voltage when the input voltage is low. This circuit is depicted in Figure 1.12. It is composed of a p-type transistor, a n-type transistor, a voltage source and a ground. The input voltage is applied to the gate of both transistors. Basically, the p-type transistor provides a low resistance between drain and source when its gate-source voltage is negative (and hence the input voltage is low) and reciprocally, it provides a high drain-source resistance when its gate-source voltage is zero. On the other hand, the n-type transistor creates a high drain-source resistance when its gate-source voltage is positive and a low resistance when its gate-source voltage is null.

Figure 1.12 shows what happens when an input is connected to both a p-type transistor and a n-type transistor. When the input voltage is low, the n-type transistor acts as a high resistance between the output and the ground. On the other hand, the drain-source resistance of the conducting p-type transistor leads to a small voltage drop between the supply voltage and the output, i.e., the output registers a voltage nearly equal to the input voltage and is thus at the high logical level. The inverse phenomenon occurs when the input voltage is high. In short, the outputs of the p-type and n-type
transistors are *complementary*, i.e., when the input is low, the output is high, and when the input is high, the output is low. This opposite behavior causes the output of this circuit to be the inversion of the input. Figure 1.13 illustrates the behavior of an *ideal* CMOS inverter.

From [43], the power dissipation of the CMOS input-inverter depicted in Figure 1.12 can be expressed as

$$P_{\text{tot}} = P_{\text{short}} + P_{\text{dyn}} + P_{\text{leak}}$$  \hspace{1cm} (1.3)

where

- $P_{\text{short}}$ is a power dissipated only while the input voltage is switching from one state (low or high voltage) to the other one.
- $P_{\text{dyn}}$ is another power dissipated only while switching the input voltage.
- $P_{\text{leak}}$ is a power dissipated while the component is in a *steady* state, i.e., while...
the input voltage remains constant.

The short circuit power dissipation $P_{\text{short}}$ is generated by a current named the through current [65] which flows from the source supply voltage to the ground while the input voltage is switching from one state (low/high voltage) to the other one (see Figure 1.14). Indeed, while the input voltage is switching, both transistors switch simultaneously from one state (blocking/non-blocking) to the other one, hence creating a path between the source $V_{dd}$ and the ground for a short period of time. During this time, a current—the through current—therefore flows from $V_{dd}$ to $V_{ss}$ and generates the power dissipation $P_{\text{short}}$ that contributes to the total power dissipation of the CMOS inverter. Notice that, for fast input transition rates, this through current is negligible compared to the switching current described below [65].

The dynamic power dissipation $P_{\text{dyn}}$. Like the short circuit power dissipation, the dynamic power dissipation is also consumed while the input voltage is switching from one
1.4 The physical layer

state to the other one. This power dissipation is generated by the switching current [65] which is due to the parasitic capacitance of the transistors—the capacitance is the ability of a component to store an electric charge. In Figure 1.14, the capacitance $C_L$ is the load capacitance and models all the parasitic capacitances of the transistors and of the connections driven by the output. When the output voltage is switching from low to high, an energy of $\frac{C_L V_{dd}^2}{2}$ is stored in $C_L$ while the same amount of energy is dissipated in the p-type transistor. When the output voltage returns to zero the energy stored in $C_L$ is dissipated in the n-type transistor. Thus, a total energy of $C_L \cdot V_{dd}^2$ is drawn from the voltage source at each pulse of the input voltage. Assuming that this NOT gate is incorporated into a processor running at frequency $f$, then in the worst-case scenario the input voltage is switched $f$ times every second. That is, an average power of $C_L \cdot V_{dd}^2 \cdot f$ is dissipated. However, this assumption following which the input voltage is switched at every clock cycle is very pessimistic and the average power dissipation is commonly

Figure 1.14: The through current flows from source to ground when both transistors are conducting.
CHAPTER 1. INTRODUCTION TO EMBEDDED REAL-TIME SYSTEMS

estimated by

\[ C_L \cdot V_{dd}^2 \cdot \beta \cdot f \]

where \( \beta \cdot f \) is the average switching frequency.

The leakage power dissipation \( P_{\text{leak}} \) is generated by the gradual loss of energy from charged capacitors (here, the two transistors) that conduct a small amount of current even when they are turned off. Basically, any current that flows when the ideal current is zero can be considered as leakage current, and unfortunately, every electronic device generates such leakage current when they are in standby, disabled, or “sleep” mode. The dominant factor of the leakage power dissipation is the subthreshold current [43]. In short, both n-type and p-type transistors have a threshold voltage \( V_{th} \) that can be defined as the minimum voltage at which the transistor turns on, i.e., the conduction of current begins—this threshold voltage can be seen as the boundary between the blocking/non-blocking state of the transistor. Even though the current (called sub-threshold current) through the transistor drops exponentially when the supply voltage goes below the threshold voltage [56], its intensity, however, is never null and contributes to the leakage consumption.

Subthreshold currents are only one actor in the leakage consumption: other leakage components that can be roughly equal in size depending on the design of the device are “gate-oxide leakage” and “junction leakage” (see [6] for details). Another contributor to the leakage consumption comes from the undesired imperfection of some dielectric materials used in capacitors, also known as dielectric leakage. Indeed, some dielectric materials are not perfect insulators and do not have a non-zero conductivity. This allows a leakage current to flow, slowly discharging the capacitor. Further technology advances using thinner gate dielectrics generate an additional leakage component, because of the current tunneling through the extremely thin gate dielectric. Fortunately, using high-k dielectrics instead of silicon dioxide that is the conventional gate dielectric allows similar device performance while avoiding this current. A good overview of leakage and reduction methods are explained in [54, 61].
1.4 The physical layer

1.4.4.4 Power dissipation of CMOS circuits: from past to present

According to Expression 1.3 and to the explanations given in the previous section, the power dissipation of a single CMOS logic gate can be estimated by $P_{\text{wr_{tot}}}$ where

$$P_{\text{wr_{tot}}} \overset{\text{def}}{=} C_L \cdot V_{dd}^2 \cdot \beta \cdot f + P_{\text{wr_{short}}} + P_{\text{wr_{leak}}} \quad (1.4)$$

where

- $C_L$ is the load capacitance,
- $V_{dd}$ and $f$ is the supply voltage and switching frequency (respectively),
- $\beta \cdot f$ is the average switching frequency,
- $f$ is the frequency of the source,
- $P_{\text{wr_{short}}}$ is the short circuit power dissipation,
- $P_{\text{wr_{leak}}}$ includes every leakage power dissipation.

Historically, CMOS circuits operated at supply voltages much larger than their threshold voltages; $V_{dd}$ might have been 5V while $V_{th}$ for both n-type and p-type transistors might have been 700 mV. Because of this high threshold voltage and because the current intensity decreases exponentially when the input voltage drops below the threshold voltage, an input voltage close to 0 generated a subthreshold current of very low intensity and thus, an insignificant power dissipation. This is the reason why former CMOS-based circuits benefited from a very low leakage and short circuit power dissipation while their dynamic power dissipation $P_{\text{wr_{dyn}}}$ was clearly dominant since most of the power was dissipated by moving charges in the logic gates. Moreover, since CMOS-based circuits (such as CMOS-based processors for instance) consist in multiple logic gates connected together, these circuits could be seen as a single “big” logic gate and, because $P_{\text{wr_{dyn}}}$ was dominant, the total power dissipation of a processor was often modeled by Expression 1.1 that only reflects the dynamic power dissipation, i.e.,

$$P_{\text{wr(t)}} \overset{\text{def}}{=} \alpha \cdot f(t) \cdot V(t)^2$$

where the factor $\alpha$ includes both the switching probabilities $\beta$ of each logic gate and an average estimation of the load capacitance $C_L$. 

53
Nowadays, manufacturers have switched to constructions that have lower supply voltage in order to reduce the dynamic power consumption and to keep electric fields inside small devices low so that the device reliability is maintained. However transistors are continuously scaled down, allowing a further reduction of the input voltage, but forcing a reduction of the threshold voltage as well. This miniaturization reduces the gap between the input voltage and threshold voltage; and because the subthreshold conduction drops exponentially with the size of this gap (the larger the gap is, the lower the conduction is), subthreshold currents becomes more and more significant as MOSFETs shrink in size [64]. A modern n-type transistor has now a threshold voltage $V_{th}$ of 200 mV (compared to 700mV in the past), resulting in a significant subthreshold leakage current. Numerous former researches (see [43, 61] for instance) advanced that continued scaling of supply voltage will make subthreshold conduction a dominate source of power dissipation and in 2005, the authors of [67] claimed that leakage consumption can even exceed 50% of the total power consumption. Actually, this inversion in the behavior of how components consume energy was not unexpected. In 2000, it was already reported in [44] that the leakage power could be larger than dynamic power in near future. Nowadays, these leakage currents are becoming a significant factor to portable device manufacturers because of their undesirable effect on battery run time for the consumer [23]. “Leakage power reduction is critical to sustaining scaling of CMOS circuits (...) understanding sources of leakage and solutions to tackle the impact of leakage will be a requirement for most circuit and system designers” [54].

1.4.4.5 Energy consumption in CMOS Circuits

As stated in the thesis title, we focus on energy rather than power dissipation. Although low-power and energy-efficiency are often perceived as overlapping goals, there are certain differences when designing for one or the other. Formally, the energy consumed by a system is the amount of power dissipated during a certain period of time:

$$E \overset{\text{def}}{=} \int_{0}^{t} \text{Pwr}(t) \cdot dt$$

Every computation, simple or complex, requires a specific interval of time to be completed. The energy consumption decreases if the time required to perform the operation decreases and/or the power consumption decreases. Thus, compared to the pure power consumption minimization problem, energy reduction includes the time aspect. A
1.4 The physical layer

A technique that would lower the power, but at the same moment increase the computational time, might even lead to an increase in energy consumption. For example, one could halve the power consumption by only halving the clock frequency in Equation 1.1 (page 45). At the same time the overall computational time required to perform the same operation would double, leading to no effect on energy consumption. On the other hand, the supply voltage forces an upper limit on the clock frequency. For this reason supply voltage and clock frequency scaling are addressed in conjunction. Note that often lower energy consumption means slower systems. Real-time scheduling and energy minimization are therefore closely related problems, that should be tackled in conjunction for best results.

In the remainder of this thesis, the part of the consumption due to the leakage power dissipation $P_{\text{leak}}$ will be referred to as “leakage consumption” while that due to the dynamic power dissipation $P_{\text{dyn}}$ will be referred to as “dynamic consumption”. Finally, we ignore the consumption due to the short circuit power dissipation $P_{\text{short}}$ because of its low intensity compared to $P_{\text{dyn}}$.

1.4.5 ASIC vs. FPGA implementation

A common configuration for very-high-volume embedded systems is the system on a chip (SoC) which contains a complete system consisting of multiple processors, multipliers, caches and interfaces on a single chip. SoCs are typically implemented as either an Application-Specific Integrated Circuit (ASIC) or using a Field-Programmable Gate Array (FPGA). These two paradigms are briefly described below.

**Application-Specific Integrated Circuit (ASIC).** It is an integrated circuit customized for a particular use, rather than intended for general-purpose use. For example, a chip designed solely to run a cell phone is typically an ASIC. As feature sizes have shrunk and design tools improved over the years, the maximum complexity (and hence functionality) possible in an ASIC has grown from 5000 gates to over 100 million. Modern ASICs often include entire 32-bit processors, memory blocks including ROM, RAM, EEPROM, Flash and other large building blocks. Such an ASIC is often termed a SoC (system-on-a-chip). Designers of digital ASICs use a hardware description language (HDL), such as Verilog or VHDL, to describe the ASIC functionalities.
Field-Programmable Gate Array (FPGA). It is an integrated circuit designed to be configured by the customer or designer after manufacturing. FPGAs are the modern-day technology for building prototypes; programmable logic blocks and programmable interconnects allow the same FPGA to be used in many different applications. The ability to update its functionality after manufacturing and to allow partial re-configuration offers advantages for many applications such as:

- shorter time-to-market because they are standard components,
- shorter development time since it reuses the basic functions and reconfigurability allows less strict prior validation,
- lower cost for small runs (typically less than 10,000 units). With technological progress, this quantity tends to increase: in fact, the price of a chip is proportional to its surface, which decreases with fine engraving, while the initial costs to build an ASIC (design, testing, masks burning) are growing rapidly.

FPGAs are used in various applications requiring digital electronics (telecommunications, aerospace, transport, etc.), but they are also used for ASIC prototyping. Indeed, FPGAs can implement any logical function that an ASIC could perform since the configuration of FPGAs is generally specified using the same hardware description languages as those used for the configuration of ASIC. However, FPGA implementations are generally slower and consume more energy than their equivalent in ASIC. Finally, many modern FPGAs have the ability to be reconfigured partially on-the-fly. This allows for reconfigurable systems—for example a CPU for which instructions change dynamically as needed.

1.4.6 Models of multiprocessor architectures

In current days, many recent embedded systems are built upon multiprocessor platforms because of their high-computational requirements. As pointed out in [16, 17], another advantage is that multiprocessor systems are more energy efficient than equally powerful uniprocessor platforms, because raising the frequency of a single processor results in a multiplicative increase of the consumption (more details in the following sections) while adding processors leads to an additive increase. Multiprocessor architectures are usually modeled by one of the three following models.
1. **Identical platform model.** In this model, the hardware specification of all the processors are assumed to be derived from the same Register Transfer Logic (RTL) description in some Hardware Description Language (HDL), such as VHDL or Verilog. Furthermore, they are assumed to use the same standard-cell library for their logic synthesis and the same technology node. Informally, the identical platform model considers that all the processors have the same characteristics, in term of consumption, computational capabilities, etc.—in any interval of time, two identical processors execute the same amount of work and consume the same amount of energy. In the remainder of this thesis, a platform composed of \( m \) identical processors will be modeled by \( \pi \) and the processors by \( P_1, P_2, \ldots, P_m \).

2. **Uniform platform model.** In this model, a parameter \( s_i \) is associated to every processor \( P_i \), with the interpretation that, in any time interval of length \( t \), processor \( P_i \) executes \( s_i \cdot t \) units of execution. This parameter can be seen as the execution speed of the processor. For instance, a job with a WCET of 10 will complete by 10 time units on a processor of speed 1 and by 5 time units on a processor of speed 2. In the remainder of this thesis, a platform composed of \( m \) uniform processors will be modeled by \( \pi = \{s_1, s_2, \ldots, s_m\} \), where \( s_i \) is the execution speed of processor \( P_i \). Without loss of generality, we assume that \( s_i \geq s_{i-1} \forall i = 2, 3, \ldots, m \), meaning that processor \( P_m \) is the fastest while processor \( P_1 \) is the lowest. Notice that this notation can also model an identical platform. Indeed, an identical platform can be modeled by \( \pi = \{s_1, s_2, \ldots, s_m\} \) with \( s_i = s_j \forall i, j \in [1, m] \).

3. **Unrelated platform model.** This model is a generalization of the uniform model. Here, a speed \( s_{ij} \) is associated to every couple processor-task \((P_i, \tau_j)\) with the interpretation that, in any time interval of length \( t \), task \( \tau_j \) executes \( s_{ij} \cdot t \) execution units when executed on processor \( P_i \). The unrelated model was introduced in order to reflect the fact that two distinct tasks (i.e., with different code-instructions) executed on the same processor can require different execution times to complete even though the length of their code is identical. This is due to internal architecture of the processors and the type of the task instructions. Indeed, some processors are optimized for some types of instructions (such as the floating point operations for instance) while they require more time to complete other types of instructions (such as the accesses to the registers).
1.5 Energy consumption of embedded real-time systems

1.5.1 Context of the problem

Nowadays, the energy consumption has become an important issue in the design of battery-powered devices. Indeed, the performance as well as the number of features of such electronic devices increase much faster than the capacity of the batteries and the demand for energy is therefore getting greater than the offer. A lot of devices in common use are suffering from this excessive demand in energy, such as laptops and mobile phones for instance. In general, most standalone devices that are autonomous in terms of energy are affected by the gap between their energy needs and their available resources. These devices use electronic components that operate at increasingly high frequencies, hence increasing their performance as well as their energy consumption (as explained in Section 1.4.4, frequency and voltage are closely connected). During the past twenty years, various techniques have been developed so that these devices can match their expected performance while minimizing their consumption. This reduction in energy consumption also allows them to be equipped with smaller batteries, making themselves smaller and lighter. Informally, power management in embedded systems are desired for many reasons, in particular: prolong the battery life-time, reduce cooling requirements and reduce operating costs for energy and cooling. Furthermore, using less energy reduces potential hazards of overheating, thereby making these devices more reliable and less dangerous [55].

1.5.2 Straightforward solutions to reduce the energy consumption

In this section, we briefly discuss some techniques used to reduce energy consumption of embedded systems, both at the hardware and software level.

In our opinion, the first step for minimizing the consumption of a system is to use a suitable hardware design. For example, if the system does not require to store a large amount of data, then it is preferable to use flash memory instead of hard drives. Indeed, physical disks consume more energy because of the mechanical process that positions their head [55]. Another solution consists in finding an appropriate trade-off between the number and the size of the memory caches. Typically, a system with a single large memory consumes more energy than a system with several smaller and distributed memories, but it consumes less time to access the data in return [55].
Some other solutions also concern only the hardware. For example, thanks to advances in the manufacturing process of electronic components, the size of these components is continuously decreased and their supply voltage decreases along with their size [55]. We worth however notice that, while modern technology contributes to reduce the size of the components, the marketing promotes adding features to the basic system (therefore adding new components) in order to stay ahead of the competition, thereby shifting the problem without solving it (for example, modern mobile phones can now take pictures and record short videos). It is also possible to reduce the energy consumption by improving the memory architecture. For instance, a cache memory can be divided into independent blocks so that only the blocks affected by the ongoing treatment are powered on. Finally, one can also cite studies about asynchronous circuits that allow to consume energy only in the parts of the system that are actually used during the execution of an instruction [55].

Among the solutions concerning the software, one can cite the possibility to optimize the source code of the application. For instance, “memory-to-memory” operations can be replaced with “register-to-register” operations in order to limit the amount of memory accesses, resulting in a lower energy consumption [55]. Moreover, since a function call is typically a lengthy and costly operation [55], some programming languages (such as “C” for instance) allows programmers to use “inlining” optimization—this method replaces every call to a function by the source code of the function itself at compilation-time, hence reducing the number of function calls. However, using inlining implies that a trade-off must be found between the execution time of the code and its size—this technique can so much increase the size of the code that it could no longer be possible to store it into a cache memory. Another strategy concerning the software is to encode information according to the binary numeral system known as “Gray code”. Following this encodement two successive instruction or data addresses differ from only one bit, hence reducing the amount of input voltage changes of the logic gates, for instance, whenever the instruction pointer flits from one instruction to the next one. This reduces the average amount $\beta$ of logic state transitions, together with the consumption of the processor (see Expression 1.4 on page 53).

Finally, there exist some hybrid techniques that involve both hardware and software features. These techniques are known as Dynamic Voltage and Frequency Scaling
CHAPTER 1. INTRODUCTION TO EMBEDDED REAL-TIME SYSTEMS

(DFVS) approaches and are discussed in the next section.

1.5.3 The Dynamic Voltage and Frequency Scaling (DVFS) framework

Among all the hardware and software investigations launched into the concern of low-power system design, the “Dynamic Voltage and Frequency Scaling” (DVFS) [26] framework has become a major issue for energy-aware computer systems. This framework consists in minimizing the energy consumption of the system by adjusting intelligently the working voltage and frequency of the processor(s). Targeting hard real-time systems, it focuses on minimizing the energy consumption while respecting all the timing constraints. This framework considers that both the RTOS and the processor(s) are designed so as to be DVFS-aware, i.e., processor(s) must allow the RTOS to modify both its supply voltage and operating frequency and the RTOS has to make an efficient use of this feature. The DVFS framework is sometimes also referred to as “DVS” (Dynamic Voltage Scaling) or “FVS” [21] (Frequency/Voltage Scaling) in the literature.

Currently, DVFS techniques provide significant results compared to conventional approaches that use energy indiscriminately. Thereby, many processors have been specifically designed to encourage the use of DVFS. Some processors even go further by providing one or several sleep modes, where each sleep mode differs from another in its energy consumption, the amount of processor parts that it turns off and the time that it needs to return to the normal (i.e., operative) mode. For instance, Intel mobile architectures such as the Intel®Pentium®M processor includes Enhanced Intel SpeedStep®Technology to optimize power and performance according to the demand on the system. This technology operates by providing multi-point operating modes (referred to as P-State, P0 being highest CPU frequency) on the CPU that increments or decrements the processor voltage and frequency depending on the demand. When there is negligible demand on the system and the CPU is idling, this technology provides multiple processor sleep states (referred to as C-State; higher C-states such as C4 refers to deeper sleep state) that reduce the overall leakage power dissipation significantly. Among the DVFS processor technologies, one can also cite Intel XScale, AMD Cool’n’Quiet, AMD PowerNow!, IBM EnergyScale and Transmeta LongRun and LongRun2. Intel XScale Technology is used by numerous popular processors, including the Intel PXA27x processor family [42] used by many PDA devices [41]. Nowadays, many computer systems, and especially embedded systems, are equipped with such voltage and frequency scaling processors and adopt various energy-efficient strategies
1.6 Outline of the thesis

As introduced in the previous section, reducing the energy consumption is a problem that can be addressed from multiple points of view. Both the hardware and software are concerned and can play a great role in the resulting consumption of the system. This is why this thesis proposes solutions that bring into play the three layers of a computer system, i.e., the application layer, the Operating System and the hardware layer. Each of the three following chapters focuses on one of these layers. Chapter 2 focuses on the application layer and investigates the particular multimode application model which is more appropriate for the energy-aware techniques proposed in Chapters 3 and 4. Chapter 3 focuses on the hardware layer and proposes a new multiprocessor hardware architecture that allows a significant reduction of the energy consumption while executing multimode applications. Finally, Chapter 4 focuses on the RTOS and presents some DVFS schedulers that further reduce the consumption of multiprocessor platforms such as the one proposed in Chapter 3. That is, Chapter 2–4 are connected together with the same objective of ultimately reducing the energy consumption of the whole system.

Chapter 2. As mentioned above, this chapter does not focus directly on the energy consumption problem in the sense that it does not propose solutions to reduce the consumption. Rather, we study a particular application model—the multimode application model—in which the application is divided into several operating modes. Each mode is in charge of its own set of functionalities, i.e., its set of tasks, with the interpretation that the application can run in only one mode at a time and running in a mode (say $M_i$) implies that only the tasks of $M_i$ are scheduled while the tasks of the other modes are disabled. As we will see in Chapters 3 and 4, modeling real-time applications by such multimode models offers a better granularity and increases the accuracy of the schedulability analyses. Besides, this application model is more appropriate to the
CHAPTER 1. INTRODUCTION TO EMBEDDED REAL-TIME SYSTEMS

energy-aware techniques proposed in Chapter 3. The problematic that we address is the following. During the execution of a multimode real-time application, switching from the current mode to another mode requires to substitute the current executing task set with the set of tasks of the destination mode. This substitution introduces a transient stage, where the tasks of the current and destination modes may be scheduled simultaneously, thereby leading to a possible overload that can compromise the system schedulability. Indeed, it can be the case that both the current and destination modes have been asserted schedulable by the schedulability analysis but the transition between them fails at run-time. Therefore, ensuring that all the deadlines are met requires not only that a schedulability test is performed on the tasks of each mode but also that (i) a protocol for transitioning from one mode to another is specified and (ii) a schedulability test for each transition is performed. In short, we propose in that chapter two protocols which ensure that all the timing requirements are fulfilled during every transition between every pair of operating modes of the application. Together with these protocols, we propose several schedulability tests which guarantee that the application can be successfully scheduled (i.e., without missing any deadline), considering different multiprocessor platform models and different scheduling policies.

Chapter 3. The main focus of this chapter is the hardware layer. We propose a new multiprocessor architecture which basically consists in two identical multiprocessor platforms placed side-by-side: one platform is composed of low-power and low-performance processors whereas the other contains high-power and high-performance processors. Together with this platform, we design appropriate algorithms which divide the set of tasks of each mode of the application into two subsets, where each subset is rigidly assigned to one of the two platforms at system design-time. The objective of the proposed algorithms is to find the partition of each task set that leads to a minimal consumption of the whole platform while running the application. In particular, we show that our new multiprocessor platform design is compatible with any processor architecture, provides significant energy savings and reduces the complex energy-aware scheduling problem to a classical 2-bins packing problem.

Chapter 4. In this chapter, we focus on the Operating System and especially on the scheduler. We target identical multiprocessor platforms and sporadic constrained-deadline tasks and we exploit the DVFS feature of the processors in order to reduce their energy consumption. For these platform and application models, we propose both offline and online DVFS schedulers. Offline algorithms, on the contrary to online
algorithms, determine the speed(s) of the processors at system design-time. Inversely, online algorithms perform “local” adjustments of the processors speed at run-time, in such a manner that the resulting speed of each processor matches its current workload as closely as possible. We demonstrate that all the algorithms that we propose do not jeopardize the schedulability of the application and we show how these techniques can be combined together to further improve the energy consumption. Finally, we perform simulations showing that all these methods can significantly reduce the energy consumption of the processors.

Conclusions and appendixes. We finally summarize the results and give concluding remarks with research perspectives in the general conclusion. Additionally, this dissertation comes with three appendixes. In Appendix A, we outline the power dissipation profile of some commercial processors. Appendix B presents additional simulation results related to Chapter 4 and finally, Appendix C presents the proofs omitted in Chapter 2.
CHAPTER 1. INTRODUCTION TO EMBEDDED REAL-TIME SYSTEMS

References


REFERENCES


CHAPTER 1. INTRODUCTION TO EMBEDDED REAL-TIME SYSTEMS


CHAPTER 1. INTRODUCTION TO EMBEDDED REAL-TIME SYSTEMS


# Algorithms for Scheduling Multimode Real-Time Applications

« Douter de tout ou tout croire sont deux solutions également commodes, qui l’une et l’autre nous dispensent de réfléchir. »

Henri Poincaré

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Introduction</td>
<td>73</td>
</tr>
<tr>
<td>2.2</td>
<td>Models of computation and specifications</td>
<td>77</td>
</tr>
<tr>
<td>2.3</td>
<td>The synchronous protocol SM-MSO</td>
<td>85</td>
</tr>
<tr>
<td>2.4</td>
<td>The asynchronous protocol AM-MSO</td>
<td>90</td>
</tr>
<tr>
<td>2.5</td>
<td>Some preliminary results for determining validity tests</td>
<td>100</td>
</tr>
<tr>
<td>2.6</td>
<td>Validity test for identical platforms and FJP schedulers</td>
<td>115</td>
</tr>
<tr>
<td>2.7</td>
<td>Validity test for identical platforms and FTP schedulers</td>
<td>125</td>
</tr>
<tr>
<td>2.8</td>
<td>Validity test for uniform platforms and FJP schedulers</td>
<td>129</td>
</tr>
<tr>
<td>2.9</td>
<td>Validity test for uniform platforms and FTP schedulers</td>
<td>166</td>
</tr>
<tr>
<td>2.10</td>
<td>Adaptation to identical platforms of the upper-bound for uniform platforms</td>
<td>171</td>
</tr>
<tr>
<td>2.11</td>
<td>Accuracy of the proposed upper-bounds</td>
<td>177</td>
</tr>
<tr>
<td>2.12</td>
<td>Simulation results</td>
<td>187</td>
</tr>
<tr>
<td>2.13</td>
<td>Validity tests at a glance</td>
<td>191</td>
</tr>
<tr>
<td>2.14</td>
<td>Conclusion and future work</td>
<td>195</td>
</tr>
</tbody>
</table>
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

Abstract

This chapter addresses the scheduling problem of multimode real-time applications upon multiprocessor platforms. During the execution of such applications, the system can switch from the current operating mode to any other mode such that the current set of tasks is replaced with that of the new-mode. Thereby, ensuring that all the deadlines are met requires not only that a schedulability test is performed on the tasks of each mode but also that (i) a protocol for transitioning from one mode to another is specified and (ii) a schedulability test for each transition is performed. We propose two protocols that ensure that all these requirements are fulfilled during every transition between every pair of operating modes of the application. The first one, named SM-MSO, is synchronous in the sense that, during any transition between two modes, it completes the execution of all the active jobs of the current mode before launching the tasks of the new-mode. The second protocol, named AM-MSO, is asynchronous with the interpretation that during the mode transitions the tasks of both the old- and new-mode are scheduled together. By extending the theory about the makespan determination problem, we formally establish a schedulability analysis for both protocols SM-MSO and AM-MSO. That is, we propose different schedulability tests that indicate beforehand whether all the timing requirements will be met during any mode transition of the system. Our study focuses first on the particular case of identical multiprocessors platforms. Then, we address the more complex issue of uniform platforms for which we will see that the solutions targeting identical platforms cannot be straightforwardly extended to uniform platforms. We provide schedulability tests for both SM-MSO and AM-MSO, assuming both identical and uniform platforms and both FTP and FJP schedulers.
2.1 Introduction

2.1.1 Motivation

As mentioned in Chapter 1, Section 1.2.3, most of real-time applications are currently modeled by a single set of tasks, where each task is characterized by a WCET, a temporal deadline and an activation rate, i.e., a period, following various interpretations such as those given in Chapter 1 (page 12). However in the real world, applications are usually designed in such a manner that they cope with all the situations generated by the environment. Therefore, applications are generally composed of several operating modes where each mode provides a dedicated behavior or serves a specific purpose, e.g., an initialization mode, an emergency mode, a fault recovery mode, etc. In such a multimode design, it is not rare that some modes contain some “unique” functionalities, i.e., tasks included in only one mode and independent from every other task.

Modeling applications in a conventional manner where the application is seen as a whole (i.e., \( \tau = \{\tau_1, \tau_2, \ldots, \tau_n\} \)) does not reflect this possible independence between the tasks. This can causes the schedulability analysis to provide very pessimistic results since such traditional analysis considers that all the tasks can be active at the same time. To illustrate this lack of precision, suppose an application \( \tau \) composed of 6 tasks \( \tau_1, \tau_2, \ldots, \tau_6 \) such that \( \sum_{i=1}^{6} U_i > 4 \). If this application is modeled by a single set of tasks, i.e., \( \tau = \{\tau_1, \tau_2, \ldots, \tau_6\} \), then \( U_{\text{sum}}(\tau) > 4 \) and \( \tau \) requires at least 5 processors in order to meet all the deadlines. Contrastingly, suppose that this application can be split into 3 distinct modes \( \tau^1 = \{\tau_1, \tau_2\} \), \( \tau^2 = \{\tau_3, \tau_4\} \) and \( \tau^3 = \{\tau_5, \tau_6\} \) such that (i) each mode is completely independent from the two others and (ii) \( U_{\text{sum}}(\tau^1) < 2 \), \( U_{\text{sum}}(\tau^2) < 2 \) and \( U_{\text{sum}}(\tau^3) < 2 \). Therefore, it could be possible that only 2 processors are required to successfully schedule \( \tau \). As a consequence, it is always preferable to model a real-time application by as many modes as possible and to perform a schedulability analysis separately on each mode, rather than performing a single analysis on the whole application. Generally, designing an application as a set of operating modes allows for a more accurate analysis that can reduce the number of required processors, which in turn reduces the energy consumption of the whole system. Furthermore, we will introduce in Chapter 3 some algorithms and hardware designs that take advantage of the fact that the application is divided into several operating modes in order to further reduce the energy consumption.
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

2.1.2 Problematic of multimode real-time applications

As introduced above, designing an application by a set of operating modes generally makes the schedulability analysis simpler and more accurate. However, it brings the following problematic. During the execution of a multimode real-time application, switching from the current mode to another one (respectively named the old-mode and the new-mode hereafter) requires to substitute the current executing task set with the set of tasks of the new-mode. This substitution introduces a transient stage, where the tasks of the old- and new-mode may be scheduled simultaneously, thereby leading to a possible overload that can compromise the system schedulability. Indeed, it can be the case that both the old- and new-mode have been asserted schedulable by the schedulability analysis but the transition between them fails at run-time.

The scheduling problem during a transition between two modes has multiple aspects, depending on the behavior and requirements of the old- and new-mode tasks when a mode change is initiated (see e.g., [27] for details about the different task requirements during mode transitions). For instance, an old-mode task may be allowed to be immediately aborted, or (on the contrary) can require to complete the execution of its current active job so that it preserves data consistency for instance. In this chapter, we assume that every old-mode task must complete its current active job (if any) when a mode change is requested, because we will see that, using scheduling algorithms such as the one considered in this chapter, tasks that can be aborted upon a mode change request do not jeopardize the schedulability of the mode transitions (we will see in Section 2.5 that it is not obvious in a multiprocessor context because of scheduling anomalies). On the other hand, a new-mode task can have two distinct requirements: either it requires to be activated as soon as possible when a mode change is requested, or it requires to be activated only when all the active jobs issued from the old-mode have completed their execution. Finally, there may be some tasks (called mode-independent tasks) present in both the old- and new-mode, such that their periodic (or sporadic) execution pattern must not be jeopardized by the mode change in progress (such tasks are typically daemon functionalities in practice). A mode-independent task is formally defined as follows.

Definition 2.1 (Mode-independent task [27])

A task is said to be mode-independent if it belongs to more than one mode.
2.1 Introduction

The transition scheduling protocols are classified with respect to (i) their ability to deal with the mode-independent tasks and (ii) the way they schedule the old- and new-mode tasks during the transitions. In the literature (see [27] for instance), the following definitions are often used. Recall that a reference is always mentioned into brackets next to the index of each result (definitions, lemmas, etc.) if the result in question has been published in the literature. Otherwise, we do not annotate the result.

**Definition 2.2 (Synchronous/asynchronous protocol [27])**

Any transition protocol is said to be synchronous if it does not schedule new-mode tasks unless all the old-mode tasks have completed. Otherwise, it is said to be asynchronous.

**Definition 2.3 (Protocol with/without periodicity [27])**

Any transition protocol is said to be “with periodicity” if it is able to deal with mode-independent tasks. Otherwise, it is said to be “without periodicity”.

In our opinion, the existing vocabulary about the multimode real-time scheduling problem is particularly deceptive. For instance, a mode-independent task is not independent of the modes but rather, it is included in more than one mode. Also, still in our opinion, the terms “with periodicity” and “without periodicity” do not reflect the fact that a given transition protocol can effectively handle mode-independent tasks. However, this vocabulary is used in the remainder of this chapter as it has been widely used in the literature.

2.1.3 Related work

Numerous transition protocols have been proposed in the literature for uniprocessor platforms (a survey of the literature in this concern is presented by the authors of [27]). Among the synchronous protocols, one can cite the *Idle Time Protocol* [28] where the periodic activations of the old-mode tasks are suspended at the first idle time-instant occurring during the transition and then, the new-mode tasks are released. The *Maximum-Period Offset Protocol* proposed in [1] is a protocol with periodicity which delays the first activation of all the new-mode tasks for a time equal to the period of the less frequent task in both modes (mode-independent tasks are not affected). The *Minimum Single Offset Protocol* in [27] completes the last activation of all the old-mode tasks and then,
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

releases the new-mode ones. This protocol exists in two versions, with and without periodicity. Concerning the asynchronous protocols, the authors of [25, 26] propose a protocol without periodicity and the authors of [29] propose a protocol with periodicity.

To the best of our knowledge, the protocols that we present in this chapter are the first and only ones that target multiprocessors platforms. Some of these results have already been published in the literature (see previous works [22, 23, 24, 30]). The scheduling problem of multimode application on multiprocessor platforms is much more complex than upon uniprocessor platform, especially due to the presence of scheduling anomalies upon multiprocessors (see Chapter 1 on page 37 for a definition). Nowadays, it is well known that real-time multiprocessor scheduling problems are typically not solved by applying straightforward extensions of techniques used for solving similar uniprocessor problems.

2.1.4 Contribution and organization

In this chapter, we propose two protocols for managing every mode transition during the execution of a multimode real-time application on uniform multiprocessor platforms. Both protocols can be considered to be an adaptation to multiprocessor platforms of the Minimal Single Offset (MSO) protocol proposed in [27]. The first one is a synchronous protocol named SM-MSO. We provide a precise description of the protocol, as well as a detailed schedulability analysis for SM-MSO without periodicity. The second protocol AM-MSO is asynchronous and without periodicity and we also provide a detailed description of the protocol, together with a schedulability analysis. In this work, we assume that every mode transition is scheduled by a global, work-conserving and FJP scheduling algorithm according to the definitions given in Section 1.3.5.3 (page 35) of Chapter 1 (the notion of work-conserving schedulers will be however refined for both cases of identical and uniform platforms). Notice that, even though we assume FJP schedulers (which include FTP schedulers), the particular case of FTP schedulers is treated separately so that the schedulability analyses of SM-MSO and AM-MSO are less generic and therefore more accurate. Our study focuses first on the particular case of identical multiprocessors platforms. Then, the more complex issue of uniform platforms is addressed and we will see that the solutions targeting identical platforms cannot be straightforwardly extended to uniform platforms.
2.2 Models of computation and specifications

The chapter is organized as follows:

- Section 2.2 defines the computational model used throughout the chapter.
- Sections 2.3 and 2.4 describe the synchronous and asynchronous protocols SM-MSO and AM-MSO, respectively.
- Section 2.5 introduces some definitions and results necessary for the establishment of our schedulability analyses for both protocols SM-MSO and AM-MSO.
- Sections 2.6–2.9 provide all the mathematical results required by the schedulability analyses of both SM-MSO and AM-MSO, assuming in turn identical platforms and FJP schedulers (in Section 2.6), identical platforms and FTP schedulers (in Section 2.7), uniform platforms and FJP schedulers (in Section 2.8) and uniform platforms and FTP schedulers (in Section 2.9).
- Section 2.10 investigates how our results for uniform platforms behave when they are applied to identical platforms. That is, we analyze the relation between these results and those designed exclusively for identical platforms.
- Section 2.11 analyzes the accuracy of our schedulability analyses.
- Section 2.12 reports on some simulation results.
- Section 2.13 summarizes the main results introduced in this chapter.
- Section 2.14 gives our conclusions and introduces our future work, together with some remaining open problems.

2.2 Models of computation and specifications

2.2.1 Application and platform model

This chapter considers multimode real-time applications. As introduced in Section 1.2.3 on page 20, any multimode real-time application \( \tau \) is composed of a set of \( x \) operating modes noted \( M^1, M^2, \ldots, M^x \) where each mode \( M^i \) has to execute the task set \( \tau^i \) composed of \( n_i \) tasks noted \( \tau^i_1, \tau^i_2, \ldots, \tau^i_{n_i} \), i.e., \( \tau^i \equiv \{ \tau^i_1, \tau^i_2, \ldots, \tau^i_{n_i} \} \). Since we do not consider mode-independent tasks in this thesis, it holds \( \forall i, j, k \) that, if \( \tau^i_k \in \tau^i \) then \( \tau^i_k \notin \tau^j \). Each task \( \tau^i_k \) is modeled by a sporadic and constrained-deadline task characterized by three parameters \( \langle C^i_k, D^i_k, T^i_k \rangle \) with the same interpretations as those introduced in Section 1.2.2, page 12.
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

Except during the transition phases, we assume that the application always runs in only one mode at a time and all the tasks are independent, i.e., there is no communication, no precedence constraint and no shared resource (except for the processors) between them. In our previous work [24], we introduced the following concept of enabled/disabled tasks.

**Definition 2.4 (Enabled/disabled tasks [24])**

At run-time, any task \( \tau^i_k \) of the application can generate jobs if and only if \( \tau^i_k \) is enabled. Symmetrically, a disabled task cannot generate jobs.

As such, disabling a task \( \tau^i_k \) prevents future job releases from \( \tau^i_k \). When all the tasks of any mode \( \tau^i \) are enabled and all the tasks of all the other modes are disabled, the application is said to be running in mode \( M^i \) (since only the tasks of mode \( \tau^i \) can release jobs). We denote the subsets of enabled and disabled tasks of \( \tau^i \) at time \( t \) by enabled(\( \tau^i, t \)) and disabled(\( \tau^i, t \)), respectively.

Concerning the platform model, we model any identical platform only by a number \( m \) of processors denoted by \( \pi_1, \pi_2, \ldots, \pi_m \). On the other hand, uniform platforms are modeled by \( \pi \overset{\text{def}}{=} [s_1, s_2, \ldots, s_m] \), where \( s_i \) denotes the computing capacity (i.e., the speed) of processor \( \pi_i \). The reader should keep in mind the definition of the processors speed that we introduced in Chapter 1 on page 56: any job executing for \( R \) time units on any processor \( \pi_i \) running at speed \( s_i \) executes \( R \cdot s_i \) execution units. We assume that the processors are indexed by non-decreasing order of processors speed, i.e., \( s_{i-1} \leq s_i \) for all \( i \in [2, m] \). That is, the fastest processor is \( \pi_m \) and the slowest one is \( \pi_1 \). Finally, \( \forall k \in [1, m] \), we denote by \( s(k) \) the cumulated speeds of the \( (m - k + 1) \) fastest processors, i.e.,

\[
s(k) \overset{\text{def}}{=} \sum_{i=k}^{m} s_i \quad (2.1)
\]

In the particular case of identical platforms, it holds \( \forall i, j \in [1, m] \) that \( s_i = s_j \) and we assume without any loss of generality that \( \forall i: s_i = 1 \).

Also, without any loss of generality, we assume for all modes \( M^i \) that \( m \leq n_i \). In Section 2.2.3, we will assume that job parallelism is forbidden and since tasks are assumed to be constrained-deadline, there are at most \( n_i \) active jobs at any time during the execution of any mode \( M^i \). As a result, it holds for any mode \( M^i \) that in every
2.2 Models of computation and specifications

schedulable system where \( m > n_i \), there are always \( m - n_i \) processors that constantly idle in a feasible schedule. We will see later that these \( m - n_i \) idling processors are the slowest ones and the problem in that case \( m > n_i \) thereby reduces to the same problem upon the subset of the \( n_i \) fastest processors among these \( m \) processors.

2.2.2 Mode transition specifications

While the application is running in any mode \( M^i \) (the old-mode), a mode change can be initiated by any task of \( \tau^i \) or by the system itself, whenever it detects a change in the environment or in its internal state for instance. This is performed by invoking a MCR\( (j) \) (i.e., a Mode Change Request), where \( M^j \) is the destination mode (the new-mode) and we denote by \( t_{\text{MCR}(j)} \) the latest invoking time of a MCR\( (j) \).

At run-time, mode transitions are managed as follows. Suppose that the application is running in mode \( M^i \) and the system (or any task) comes to request a mode change to mode \( M^j \), with \( j \neq i \). At time \( t_{\text{MCR}(j)} \), the system entrusts the scheduling decisions to a transition protocol. This protocol immediately disables all the old-mode tasks, thus preventing them from releasing new jobs. At this time, the active jobs issued from these disabled tasks, henceforth called the rem-jobs (for “remaining jobs”), may have two distinct behaviors: either they can be aborted upon the MCR or they must complete their execution. Aborting a job consists in suddenly stopping its execution and removing it from the system memory. But in the real world, suddenly killing a process may cause system failures and the rem-jobs often have to complete their execution (to preserve data consistency for instance). From a schedulability point of view, we will show in this chapter that aborting some (or all) rem-jobs upon a mode change request does not jeopardize the system schedulability during the transition phase. Consequently, in the remainder of this chapter, we assume the worst-case scenario for every mode transition, i.e., the scenario in which every old-mode task has to complete its last released job (if any) during every mode change.

The fact that the rem-jobs have to complete upon a MCR brings the following problem. Even if both task sets \( \tau^i \) and \( \tau^j \) have been asserted to be schedulable upon the \( m \) processors at system design-time, the presence of the rem-jobs may cause an overload during the transition phase (at run-time) if all the new-mode tasks of \( \tau^j \) are enabled immediately upon the mode change request. Indeed, the schedulability analysis
performed on $\tau^j$ at system design-time did not take into account the additional work provided by the rem-jobs. To solve this problem, transition protocols usually delay the enablement of each new-mode task until it is safe to enable it. But these delays on the enablement of the new-mode tasks are also subject to hard constraints. More precisely, we denote by $D^j_k(M')$ the relative mode change deadline of the task $\tau^j_k$ during every transition from the mode $M'$ to the mode $M^j$, with the following interpretation: the transition protocol must ensure that $\tau^j_k$ is enabled not after time $t_{MCR(j)} + D^j_k(M')$. Finally, when all the rem-jobs are completed and all the new-mode tasks of $\tau^j$ are enabled, the system entrusts the scheduling decisions to the scheduler $S^j$ of the new-mode $M^j$ and the transition phase ends. In short, the goal of any transition protocol is to fulfill the following during every mode change:

1. Complete every rem-job $\tau^j_{a,b}$ by its absolute deadline $d^j_{a,b}$.
2. Enable every new-mode task $\tau^j_k$ by its absolute mode change deadline $t_{MCR(j)} + D^j_k(M')$.
3. Complete every new-mode jobs $\tau^j_{a,b}$ by its absolute deadline $d^j_{a,b}$. This concerns only asynchronous protocols as synchronous protocols enable all the new-mode tasks only at the very end of the transitions, i.e., synchronous protocols do not schedule the new-mode tasks during the transition phases.

**Definition 2.5 (Valid protocol – from our previous work [24])**

A transition protocol $P$ is said to be valid for a given application $\tau$ and platform $\pi$ if and only if $P$ enables to meet all the job and mode change deadlines during every transition between every pair of operating modes of $\tau$.

This notion of “valid protocol” is directly related to a “validity test” as defined below.

**Definition 2.6 (Validity test – from our previous work [24])**

For a given transition protocol $P$, a validity test is a condition based on the tasks and platform characteristics that indicates a priori whether $P$ is valid for a given application $\tau$ and platform $\pi$.

Recall that in the scope of this thesis, we do not consider multimode real-time applications with mode-independent tasks.
2.2 Models of computation and specifications

2.2.3 Scheduler specifications

We consider in this chapter the global scheduling problem of sporadic and constrained-deadlines tasks on multiprocessor platforms. Recall that “global” scheduling algorithms, on the contrary to partitioned algorithms, allow different tasks and different jobs of the same task to be executed upon different processors. We consider that every mode $M^k$ uses its own scheduler denoted by $S^k$ which can be either Fixed-Task-Priority (FTP) or Fixed-Job-Priority (FJP). Recall that FTP schedulers assign a priority to each task at system design-time (i.e., before the execution of the application) and then at run-time, every released job uses the priority of its task. On the other hand, FJP schedulers determine the priority of every job at run-time, where different jobs issued from the same task may have different priorities\(^1\). For both FTP and FJP schedulers, the priority of every job never changes between its release and its completion time. Furthermore, we assume that the priority of any active job is distinct from that of any other active job, i.e., at any time, two active jobs cannot have the same priority.

The scheduler of every mode is assumed to be work-conserving but, instead of using the definition given on page 35 which encompasses a whole family of scheduling disciplines, we introduce here two refinements of this concept that we name weakly and strongly work-conserving schedulers, respectively. These two definitions refine the scheduling rules when two (or more) processors are available for the execution of a waiting job. The objective by introducing these two refinements is to have one and only one possible schedule for any given set of synchronous jobs, any given platform and any given job priority assignment.

**Definition 2.7 (Weakly work-conserving scheduler)**

A scheduler $S$ is said to be weakly work-conserving if and only if it satisfies the following conditions.

- No processor idles while there are active jobs awaiting execution.

\(^1\)According to these interpretations, FTP schedulers are a particular case of FJP schedulers in which all the jobs issued from a same task receive the same priority which is determined beforehand.
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

- If a subset $P$ of processors running at the same speed idle (or complete a job) simultaneously at time $t$ then $S$ assigns the highest priority waiting job (if any) upon the highest indexed processor in $P$ (i.e., the fastest processor of $P$ according to our processor indexation rule). Informally, when the highest-priority waiting job $\tau_{i,j}$ is about to be dispatched, if more than one processor idle (and are thus available for the execution of $\tau_{i,j}$) then $\tau_{i,j}$ is dispatched to the idling processor with the highest index.

**Definition 2.8 (Strongly work-conserving scheduler)**

A scheduler $S$ is said to be strongly work-conserving if and only if it satisfies the following conditions.

- No processor idles while there are active jobs awaiting execution.
- At every time during the system execution, the job-processor assignment uses the rule: highest priority active job upon highest indexed processor.

Notice that the refinement of “weakly” work-conserving scheduler can play a role only when the highest-priority waiting job has to be dispatched. On the contrary, the refinement of “strongly” work-conserving scheduler can play a role at any time during the system execution. It is essential to keep in mind that in our study, weakly work-conserving schedulers will be used only on identical platforms whereas strongly work-conserving schedulers will be used only on uniform (and non-identical) platforms. For strongly work-conserving schedulers, the concept of migrating jobs to faster processors as soon as possible (as specified by the second condition of Definition 2.8) has been commonly used over the years in the literature about uniform platforms (see for instance [7, 8, 9, 12, 13, 15]). The definition of these two refinements is extremely important for the remainder of this chapter, especially because of their resulting properties listed below.

- Upon identical multiprocessor platforms: for any finite set $J$ of jobs scheduled by any weakly work-conserving scheduler $S$, there is one and only one possible schedule of $J$. This property is illustrated in Figure 2.1 where 5 jobs $J_1, J_2, J_3, J_4, J_5$ of respective processing time 4, 8, 8, 4, 6 are scheduled upon 2 identical processors.
2.2 Models of computation and specifications

according to a global and FJP scheduler $S$ following which $J_1 > J_2 > J_3 > J_4 > J_5$. If $S$ is work-conserving but not weakly work-conserving then Figures 2.1a and 2.1b depict two possible schedules of $J$. Otherwise, if $S$ is weakly work-conserving then Figure 2.1c depicts the only possible schedule of $J$. At time 0, processors $\pi_1$ and $\pi_2$ idle and the second condition of the definition of weakly work-conserving schedulers imposes on $J_1$ to execute on $\pi_2$. From the same rule, $J_5$ must execute on $\pi_2$ at time 12.

Upon uniform multiprocessor platforms: for any finite set $J$ of jobs scheduled by any strongly work-conserving scheduler $S$, the schedule of $J$ forms a staircase (see Figure 2.2). Indeed, since the processors are indexed in such a manner that $s_i \geq s_j \forall i > j$, it holds from the second condition of the definition of a strongly work-conserving scheduler that at any instant $t$, if $S$ idles the $i^{th}$-slowest processor then $S$ also idles the $j^{th}$ slowest processors for all $j < i$. Also, it results from the same condition that the $i^{th}$ processor that starts idling is always the processor $\pi_i$.

In the remainder of this chapter, we use the notation $\mathcal{P}$ to refer to a specific job priority assignment. A job priority assignment can be seen as a key component of any scheduler, but the definition of a scheduler is more general since, in addition to a job priority assignment, a scheduler must also specifies some specifications like “global or partitioned”, “preemptive or non-preemptive”, etc. For any job priority assignment $\mathcal{P}$, we denote by $J_i >_P J_j$ the fact that job $J_i$ has a higher priority than $J_j$ according to $\mathcal{P}$, and we assume that every assigned priority is distinct from the others. That is, $\forall \mathcal{P}, i, j$ such that $i \neq j$ we have either $J_i >_P J_j$ or $J_i <_P J_j$. Similarly, and without any distinction with the interpretation given above, we will sometimes use the notations $J_i >_{S^k} J_j$ and $J_i <_{S^k} J_j$ where $S^k$ is the scheduler of mode $M^k$, and we will sometimes use the notations $J_i > J_j$ and $J_i < J_j$ when the job priority assignment has no label (for instance, when we will depict some examples of schedules, we will just say “$J_i > J_j$” without giving a name to the job priority assignment).

Finally, the problems and solutions presented in this chapter are addressed under the following assumptions:

1. Every task set $\tau^k$ is schedulable by the scheduler $S^k$ on the $m$-processor platform. This hypothesis allows for focusing only on the schedulability of the tasks during the transient phases corresponding to mode transitions, rather than on the schedulability of the application during the execution of its modes.
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

(a) A possible schedule of $J_1$, $J_2$, $J_3$, $J_4$ and $J_5$ without using the refinement of weakly work-conserving scheduler.

(b) Another possible schedule of $J_1$, $J_2$, $J_3$, $J_4$ and $J_5$ without using the refinement of weakly work-conserving scheduler.

(c) The only possible schedule of $J_1$, $J_2$, $J_3$, $J_4$ and $J_5$ with the refinement of weakly work-conserving scheduler.

Figure 2.1: Thanks to our definition of weakly work-conserving schedulers, only one schedule is possible for a given set of jobs, an identical platform and a specific priority assignment.
2.3 The synchronous protocol SM-MSO

2.3.1 Description of the protocol

The protocol SM-MSO (which stands for “Synchronous Multiprocessor Minimum Single Offset” protocol) is an extension to multiprocessor platforms of the protocol MSO, defined in [27] for uniprocessor platforms only. The main idea of SM-MSO is the following: upon a MCR(j), all the tasks of the old-mode (say M') are disabled and the rem-jobs continue to be scheduled by the old-mode scheduler $S^i$ upon the $m$ processors.

2. Job migrations are permitted and are carried out with no loss or penalty.

3. Job parallelism is forbidden, i.e., jobs can execute on at most one processor at any instant in time.

Figure 2.2: For any fixed set of jobs and any uniform platform, the schedule generated by any strongly work-conserving scheduler forms a staircase. This phenomenon will be sometimes referred as the “staircase property” in the remainder of this chapter.

2.3 The synchronous protocol SM-MSO
Once all the rem-jobs are completed, all the new-mode tasks (i.e., the tasks of $\tau^j$) are simultaneously enabled. Algorithm 2 gives the pseudo-code of this protocol.

**Algorithm 2: SM-MSO protocol**

Input:
- $M^i$: the old mode;
- $M^j$: the new-mode

begin

| During the whole transition phase: |
| Schedule all the rem-jobs according to the old-mode scheduler $S^i$ ; |
| end procedure |

| At the completion of any rem-job $I_k$ at time $t$: |
| if (active($\tau^i$, $t$) = $\phi$) then |
| enable all the new-mode tasks of $\tau^j$ ; |
| enter the new-mode $M^j$ ; |
| end if |
| end procedure |

end

**Figure 2.3:** Illustration of a mode transition handled by SM-MSO.
2.3 The synchronous protocol SM-MSO

Figure 2.3 illustrates how SM-MSO handles the mode transitions. For sake of pedagogy, this example considers a very particular case: the platform is composed of only 2 identical processors and the application is composed of 2 modes $M^i$ and $M^j$ depicted in blue and red, respectively. These two modes contain only periodic, synchronous and implicit-deadline tasks. The blue mode $M^i$ contains 4 tasks $\tau_{i1}, \tau_{i2}, \tau_{i3}, \tau_{i4}$ and uses a FTP scheduler $S^i$ such that $\tau_{i1} >_{S^i} \tau_{i2} >_{S^i} \tau_{i3} >_{S^i} \tau_{i4}$. The parameters of these tasks are: $C_{i1} = 40, C_{i2} = 20, C_{i3} = 60$ and $D_{ik} = T_{ik} = 120 \forall k = 1, 2, 3, 4$. The red mode $M^j$ contains 3 tasks $\tau_{j1}, \tau_{j2}, \tau_{j3}$ and uses a FTP scheduler $S^j$ following which $\tau_{j1} >_{S^j} \tau_{j2} >_{S^j} \tau_{j3}$. The parameters of these tasks are: $C_{j1} = 100$ and $C_{j2} = C_{j3} = 40$. The deadline and period of these new-mode tasks do not have any importance in this example and we voluntarily omitted to specify them. At time 0, every old-mode task releases its first job and these four jobs are scheduled on the two processors according to $S^i$. At time 100, all these jobs are completed and both CPUs idle until time 120. At this time 120, every old-mode task of $M^i$ releases its second job and the scheduler $S^i$ starts the execution of $\tau_{1,2}$ and $\tau_{2,2}$ on processor $\pi_2$ and $\pi_1$, respectively. Then, the system requests a mode change at time 130. Here starts the transition phase to mode $M^j$. As specified by the protocol SM-MSO, all the old-mode tasks are immediately disabled and SM-MSO continues to schedule the active jobs $\tau_{i1,2}, \tau_{i2,2}, \tau_{i3,2}$ and $\tau_{i4,2}$ (named the rem-jobs from this point forward) according to the old-mode scheduler $S^i$. These rem-jobs execute until time 220, time at which they are all completed. At this instant 220, the condition at line 8 of Algorithm 2 is verified. Thus, SM-MSO enables all the new-mode tasks (the red ones) simultaneously and starts scheduling the released new-mode jobs according to the new-mode scheduler $S^j$. Notice that at any time during any transition phase, our protocol SM-MSO allows the system (or any task) to request any other mode change. At the very end of any transition phase (for instance, at time 220 in our example), SM-MSO enables all the tasks of the mode $M^z$ if MCR($z$) is the last mode change that has been requested.

2.3.2 Main idea of the validity test

In order to establish a validity test for the protocol SM-MSO, two key results are required:

1. First, it must be shown that disabling the old-mode tasks upon a MCR does not jeopardize the schedulability of the rem-jobs when they continue to be scheduled by the old-mode scheduler. That is, it must be guaranteed that the absolute deadline $d_{a,b}^i$ of every rem-job $\tau_{a,b}^i$ is met during every mode transition from every mode $M^i$. 

87
2. Second, it must be proved for every mode transition that the length of the transition phase can never be larger than the minimum mode change deadline of the new-mode tasks. Indeed, it follows from this statement and the definition of SM-MSO that all the mode change deadlines are met during every mode transition.

We showed the first key result mentioned above in [24] (the proof is repeated in Section 2.5, page 103), and this result holds for any uniform platform (thus including identical platforms). About the second key result, we worth notice that there is no job release (and therefore no preemption) during every transition phase since we consider only FJP schedulers and since the old-mode tasks are disabled upon every mode change request. As a consequence, the length of every transition phase corresponds to the time needed to complete all the rem-jobs (this fact clearly appears in Figure 2.3). In the literature (and hereafter), the time needed to complete a given set of jobs (all ready at a same time) upon a given multiprocessor platform is called the makespan. A more formal definition adapted to the scope of our study is given below.

**Definition 2.9 (Makespan)**

Consider the following notations:

- $J = \{J_1, J_2, \ldots, J_n\}$ denotes any set of $n$ jobs of processing times $c_1, c_2, \ldots, c_n$,
- $\pi$ denotes any uniform multiprocessor platform composed of $m$ processors (including identical platforms), and
- $S$ denotes the schedule of $J$ upon $\pi$ while using any work-conserving scheduler (including weakly and strongly work-conserving schedulers).

We define the makespan as the earliest instant in $S$ such that the $n$ jobs of $J$ are completed.

Obviously, the value of the makespan depends on the number and processing times of the jobs in $J$ (as well as on the processor speeds). That is, for a given platform, the length of any transition phase from any mode $M^i$ to any other mode $M^j$ depends on both the number and the remaining processing time of the rem-jobs at time $t_{MCR(j)}$. As a consequence, determining an upper-bound on the makespan for every transition from a given mode $M^i$ to another given mode $M^j$ requires to consider the worst-case scenario, i.e., the scenario in which the number and the remaining processing time of the rem-jobs at time $t_{MCR(j)}$ is such that the generated makespan is maximum. The worst-case
scenario is thus entirely defined by a specific set of rem-jobs that we name the \textit{worst-case rem-job set}. This notion is defined as follows.

\textbf{Definition 2.10 (Worst-case rem-job set $J^{wc}_i$)}

Assuming any transition from a specific mode $M^i$ to any other mode $M^j$, the worst-case rem-job set $J^{wc}_i$ is the set of jobs issued from the tasks of $\tau^i$ that leads to the largest makespan.

For any work-conserving FJP scheduler (including FTP schedulers) and any uniform platform (including identical platform), we will show that the worst-case rem-job set $J^{wc}_i$ of every transition from mode $M^i$ to mode $M^j$ is the one where each task $\tau^i_k$ has a rem-job at time $t_{MCR(j)}$ with a remaining processing time equals to $C^i_k$ (recall that $C^i_k$ is the WCET of $\tau^i_k$). This result is very intuitive: the makespan is as large as the number and processing times of the rem-jobs are large.

In this chapter, we address the problem of establishing mathematical expressions that provide the \textit{maximum makespan} for any given set of synchronous jobs (especially for the worst-case rem-job set of each mode transition). This intention stems from the fact that the knowledge of the maximum makespan allows to assert (or refute) that every new-mode task will meet its mode change deadline during any mode transition using SM-MSO. That is, this allows for ensuring the validity of SM-MSO for a given application $\tau$ and platform $\pi$ as follows.

\textbf{Validity Test 2.1 (For protocol SM-MSO)}

For any multimode real-time application $\tau$ and any uniform multiprocessor platform $\pi$, protocol SM-MSO is valid provided that, for every mode $M^i$,

$$\overline{ms}(J^{wc}_i, \pi) \leq \min_{j \neq i} \left\{ \min_{1 \leq k \leq n_j} \left\{ D^j_k(M^j) \right\} \right\}$$

where $\overline{ms}(J^{wc}_i, \pi)$ is an upper-bound on the makespan that could be produced during any transition from mode $M^i$.

\footnote{During every mode transition, the considered jobs are assumed to be synchronous (i.e., ready to execute at time 0) because every rem-job is active and ready to execute upon the mode change request.}
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

This validity test is a sufficient condition that indicates, a priori, if all the deadlines will be met during all possible mode changes using the protocol SM-MSO. Unfortunately, to the best of our knowledge, the problem of determining the maximum makespan has never been studied in the literature. Rather, authors usually address the problem of determining a job priority assignment that minimizes the makespan (the goal of these studies is to ultimately reduce the completion times of the jobs as much as possible). This problem of finding priorities that minimize the makespan can be cast as a strongly NP-hard bin-packing problem [14, 16] for which numerous heuristics have been proposed in the literature. In the scope of our study, Sections 2.6–2.9 provides upper-bounds on the makespan, assuming identical and uniform platforms as well as FJP and FTP schedulers.

2.4 The asynchronous protocol AM-MSO

2.4.1 Description of the protocol

The protocol AM-MSO (which stands for “Asynchronous Multiprocessor Minimum Single Offset” protocol) is an asynchronous version of SM-MSO described in the previous section. The main idea of this second protocol is to reduce the delay applied to the enablement of the new-mode tasks, by enabling them as soon as possible. On the contrary to SM-MSO, rem-jobs and new-mode tasks can be scheduled simultaneously during the transition phases according to the scheduler $S^{\text{trans}}$ defined as follows: (i) the priorities of the rem-jobs are assigned according to the old-mode scheduler; (ii) the priorities of the jobs issued from the new-mode tasks are assigned according to the new-mode scheduler, and (iii) the priority of each rem-job is higher than the priority of every new-mode job. Formally, suppose that the system is transitioning from mode $M^{\text{old}}$ to mode $M^{\text{new}}$ and let $J_i$ and $J_j$ be two active jobs during this mode change. According to these notations we have

\[
J_j >_{S^{\text{trans}}} J_i \quad \text{iff} \quad \begin{cases} 
(J_j \in M^{\text{old}} \text{ and } J_i \in M^{\text{new}}) \\
\text{or } (J_j \in M^{\text{old}} \text{ and } J_i \in M^{\text{old}} \text{ and } J_j >_{S^{\text{old}}} J_i) \\
\text{or } (J_j \in M^{\text{new}} \text{ and } J_i \in M^{\text{new}} \text{ and } J_j >_{S^{\text{new}}} J_i)
\end{cases}
\]
The main idea of AM-MSO is the following: upon a MCR($j$), all the old-mode tasks are disabled and the rem-jobs continue to be scheduled by $S^{\text{old}}$ (assuming that $M^{\text{old}}$ is the old-mode). Whenever any rem-job completes (say at time $t$), at least one processor becomes idle if there is no more waiting rem-jobs at time $t$. In that case, AM-MSO immediately enables some new-mode tasks, on the contrary to SM-MSO which waits for the completion of all the rem-jobs. In order to select the new-mode tasks to enable at time $t$, AM-MSO uses the following heuristic: it considers every disabled new-mode task by non-decreasing order of mode change deadline and it enables those which can be scheduled by $S^{\text{new}}$ upon the current available CPUs, i.e., the CPUs that are not running a rem-job and are therefore available for executing some new-mode tasks.

![Figure 2.4](image)

**Figure 2.4:** Illustration of a mode transition handled by AM-MSO.

Figure 2.4 depicts an example on a 2-processors platform, assuming the same task sets as in Figure 2.3. At time $t$ during the mode transition, the rem-job $\tau^{i}_{3,2}$ completes on processor $\pi_1$ and there is no more waiting rem-jobs. At that time, AM-MSO scans every disabled task of $\tau^{j}$ (in non-decreasing order of mode change deadline) and enables some of them in such a manner that the resulting set of enabled new-mode tasks can be scheduled by $S^{j}$ upon 1 processor (since at this time $t$, only the processor $\pi_1$ is available for executing some new-mode tasks). We have no guarantee that scanning all the disabled tasks in non-decreasing order of mode change deadline is optimal.
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

Algorithm 3: AM-MSO protocol

Input:
- $M^i$: the old mode;
- $M^j$: the new-mode

begin

Initialization phase (upon the mode change request, say at time $t$):
- Disable all the tasks of $\tau^i$;
- Sort the task set “disabled($\tau^i$, $t$)” by non-decreasing order of mode change deadlines;
- $\pi^\text{old} \leftarrow \pi$;
- $\pi^\text{new} \leftarrow \emptyset$;

end procedure

During the whole transition phase:
- Schedule all the rem-jobs and new-mode jobs according to $S^\text{trans}$;

end procedure

At the completion of any job $\tau_{k,x}$ on any processor $\pi_\ell$ at time $t$:
- if ($\tau_k \in \tau^i$ and wait($\tau^i$, $t$) = $\emptyset$) then
  - $\pi^\text{old} \leftarrow \pi^\text{old} \setminus \{\pi_\ell\}$
  - $\pi^\text{new} \leftarrow \pi^\text{new} \cup \{\pi_\ell\}$
  - foreach $\tau^j \in \text{disabled}(\tau^i, t)$ do
    - $\tau^\text{temp} \leftarrow \text{enabled}(\tau^i, t) \cup \{\tau^j\}$
    - if (sched($\pi^\text{new}$, $S^j$, $\tau^\text{temp}$)) then enable $\tau^j$
  - end foreach
- if (active($\tau^i$, $t$) = $\emptyset$) then enter the new-mode $M^j$;

end procedure

but this heuristic appears as the most intuitive choice. Notice that, on the contrary to SM-MSO, the protocol AM-MSO does not allow the system to request mode changes at any time during the transition phases. More precisely, AM-MSO allows mode changes to be requested during the mode transitions only until some new-mode tasks have been enabled (the instant $t$ in Figure 2.4). Indeed, if the system is transitioning from any mode $M^i$ to any other mode $M^j$ and a mode change is requested to any mode $M^z$ before that time, AM-MSO can then consider that the system is transitioning from mode $M^i$ to mode $M^z$ and the new-mode therefore becomes the mode $M^z$. After that time $t$, some tasks of mode $M^j$ have already been enabled and AM-MSO does not allow the system
2.4 The asynchronous protocol AM-MSO

to request any other mode change until the end of the transition phase, i.e., until all the
tasks of mode $M^l$ are enabled.

In order to determine whether a task can be safely enabled, protocol AM-MSO uses
a binary function $\text{sched}(\pi, S, \tau^l)$ defined as follows.

$$\text{sched}(\pi, S, \tau^l) \equiv \begin{cases} 
\text{True} & \text{if the task set } \tau^l \text{ is schedulable by } S \text{ upon } \pi; \\
\text{False} & \text{otherwise.} 
\end{cases}$$

This function is useful as we must always guarantee that all the deadlines are met for all
the jobs in the system, including the deadlines of all the new-mode jobs. Considering a
specific scheduler $S$, such a function can be derived from schedulability tests proposed
for $S$ in the literature\(^1\). At time 220, AM-MSO performs the same treatment as at time $t$.
But since we assumed that every task set $\tau^k$ is schedulable by $S^k$ on $\pi$, we know that all
the remaining disabled new-mode tasks can be enabled at this time 220. Algorithm 3
gives a pseudo-code of the protocol AM-MSO.

2.4.2 Main idea of the validity test

For a given application $\tau$ and platform $\pi$, the main idea to determine whether AM-MSO
allows to meet all the mode change deadlines is to run Algorithm 3 for every possible
mode transition, while considering the worst-case scenario for each one—the scenario
in which the new-mode tasks are enabled as late as possible. From our definition of
protocol AM-MSO, we know that every instant at which some new-mode tasks are
enabled corresponds to an instant at which at least one processor has no more rem-job
to execute, i.e., an “idle-instant” defined as follows.

---

\(^1\)To the best of our knowledge, there is no efficient necessary and sufficient schedulability test for any
multiprocessor scheduler that complies with the requirements specified in Section 2.2.3. Theodore Baker
has proposed in [20] a necessary and sufficient schedulability test for arbitrary-deadline sporadic tasks
scheduled by Global-EDF but its time-complexity is very high so only small applications can be tested.
Fortunately, many sufficient schedulability tests have been proposed for scheduler such as Global-EDF
(see for instance [3, 4, 5, 8, 11]) and Global-DM (see for instance [2, 6, 7]).
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

Definition 2.11 (Idle-instant idle\(k\)(\(J, \pi, P\)))

Let \(J = \{J_1, J_2, \ldots, J_n\}\) be any finite set of \(n\) synchronous jobs. Let \(\pi\) be a multiprocessor uniform platform and let \(P\) be the job priority assignment used during the schedule of \(J\) upon \(\pi\). If \(S\) denotes that schedule then the idle-instant \(\text{idle}_k(J, \pi, P)\) (with \(k = 1, \ldots, m\)) is the earliest instant in \(S\) such that at least \(k\) processors idle. Notice that \(\text{idle}_k(J, \pi, P)\) can be null.

To familiarize the reader with this notion of “idle-instants”, Figure 2.5 depicts these instants through a simple example. In this example, a set \(J\) of 7 synchronous jobs are scheduled upon a platform \(\pi\) composed of 4 identical processors according to a weakly work-conserving scheduler with the job priority assignment \(P\) following which \(J_1 \succ_P J_2 \succ_P \cdots \succ_P J_7\). From the definition of the idle-instants, it is clear that the makespan corresponds to the idle-instant \(\text{idle}_m(J, \pi, P)\) (here, \(\text{idle}_4(J, \pi, P)\)).

By definition of the protocol AM-MSO, and in particular by definition of \(S^{\text{trans}}\), a new-mode job never preempts a rem-job during the transition phases. Thereby, during every mode change, new-mode tasks are enabled at each idle-instant \(\text{idle}_k(J, \pi, P)\) \((\forall k = 1, \ldots, m)\) where \(J\) is the set of rem-jobs at the MCR occurring time and \(P\) is the job priority assignment derived from the old-mode scheduler when the mode change is requested. For obvious reasons, the exact values of these idle-instants depend on both the number of jobs in \(J\) and their actual execution times. Therefore, these exact value cannot be determined at system design-time and the main idea of our validity test is the following. First, for every mode \(M^i\) we determine the set \(J\) of rem-jobs that leads to the largest idle-instants \(\text{idle}_k(J, \pi, P)\) \((\forall k \in [1, m])\). From this point forward, we thus refine the definition of the worst-case rem-job set as follows.

Definition 2.12 (Worst-case rem-job set \(J^{\text{wc}}_i\))

Assuming any transition from a specific mode \(M^i\) to any other mode \(M^j\), the worst-case rem-job set \(J^{\text{wc}}_i\) is the set of jobs issued from the tasks of \(\tau^i\) that leads to the largest idle-instants.

As it will be shown in Lemma 2.1 (page 107), the worst-case rem-job set \(J^{\text{wc}}_i\) of every mode \(M^i\) is the one that contains one job \(J^i\) for each task \(\tau^i\) and such that every job \(J^i \in J^{\text{wc}}_i\) has a processing time equals to \(C^i\), i.e., the WCET of the task \(\tau^i\). Infor-
mally, the worst-case scenario during any mode transition is the one in which (i) every old-mode task releases a job exactly when the mode change is requested and (ii) every released job executes for its WCET. Second, we determine (for any given set \( J \) of jobs) an upper-bound \( \text{idle}_k(J, \pi, \mathcal{P}) \) on each idle-instant \( \text{idle}_k(J, \pi, \mathcal{P}) \) (for \( k = 1, 2, \ldots, m \)). Finally, we simulate Algorithm 3 at each instant \( \text{idle}_k(J_{wc}, \pi, \mathcal{P}) \). That is, we verify whether all the mode change deadlines are met while enabling the new-mode tasks at each instant \( \text{idle}_k(J_{wc}, \pi, \mathcal{P}) \) (following the same process as that of Algorithm 3). Obviously, if every mode change deadline is met during this simulation then every mode change deadline will be met during the actual execution of the application.

As mentioned above, a first step toward the determination of the upper-bounds \( \text{idle}_k(J, \pi, \mathcal{P}) \) (\( \forall k \in [1, m] \)) is to observe that they can be determined by considering only the schedule of the rem-jobs. Indeed, the new-mode tasks that are enabled during the

![Figure 2.5: Illustration of the idle-instants.](image-url)
transitions do not interfere with the schedule of the rem-jobs since $S^{\text{trans}}$ assigns them a lower priority than that of every rem-job. As we will see later, FTP schedulers allow for determining at system design-time a tight upper-bound $\overline{\text{idle}}_k(J, \pi, \mathcal{P})$ on each idle-instant $\text{idle}_k(J, \pi, \mathcal{P})$ (for all $k = 1, 2, \ldots, m$) during every mode transition, because:

1. the job priority assignment $\mathcal{P}$ is known beforehand and
2. even though the actual set of rem-jobs is not known beforehand, basing the computation on the worst-case rem-job set leads to maximum idle-instants that can actually be reached if every rem-job executes for its WCET.

On the other hand, FJP schedulers lead to much more pessimistic results because, for any mode transition, the job priority assignment $\mathcal{P}$ is not known beforehand. Thus, all the job priority assignments have to be considered in order to bound the idle-instants from above. At first blush, assuming that $\mathcal{P}$ is unknown can seem inconsistent since during every mode transition we consider the worst-case rem job set in the computation of each upper-bound $\overline{\text{idle}}_k(J, \pi, \mathcal{P})$ (and this worst-case rem-job set is determined at system design-time). Therefore, it could be thought that $\mathcal{P}$ can simply be derived from this worst-case rem-job set. But this intuition is erroneous because, for a given FJP scheduler, several job priority assignments can be derived from the same worst-case rem-job set. This is shown in the following Lemma 2.1.

**Lemma 2.1**

It is always possible to derive more than one job priority assignment from the same worst-case rem-job set $J_i^{\text{wc}}$.

**Proof**

The proof is made by providing a counterexample. For sake of pedagogy, this counterexample considers an identical platform but the lemma can be easily extended to uniform platforms. Let $\pi$ be an identical platform composed of 2 processors and let $\tau$ be a multimode real-time application. Suppose a mode change from mode $M_i$ to mode $M_j$ and suppose that the old-mode scheduler $S_i$ is EDF. Again, one can easily find other counterexamples with different scheduling policies. The mode $M_i$ is composed of three tasks $\tau^1_i, \tau^2_i$ and $\tau^3_i$ such that $C^1_i = 5, C^2_i = 5, C^3_i = 7$ and $D^1_i = T^1_i = 15, D^2_i = T^2_i = 16$ and $D^3_i = T^3_i = 18$. As introduced earlier, the worst-case rem-job set $J_i^{\text{wc}}$ for this mode transition is composed of three jobs $\{J_1, J_2, J_3\}$ of respective processing time $C^1_i, C^2_i$ and $C^3_i$. This will be formally proved in Corollary 2.1 on page 107, assuming any FJP scheduler and any uniform platform.
In other words, this set \( \mathcal{J}_{\text{wc}} \) of jobs leads to the maximum idle-instants (by definition of the worst-case rem-job set). But this worst-case rem-job set specifies only the processing time of the jobs, and not their release time and absolute deadline. Consequently, different job priority assignments can be derived from \( \mathcal{J}_{\text{wc}} \) and we depict two of them in Figure 2.6. In this figure, the time is relative to the instant \( t_{\text{MCR}}(j) \) (i.e., \( t_{\text{MCR}}(j) = 0 \)). The release time and the absolute deadline of each job \( j_k \) is denoted by \( a_k \) and \( d_k \), respectively. These two job priority assignments are obtained as follows.

1. If we assume that the three jobs are released exactly at the MCR occurring time \( t_{\text{MCR}}(j) \), i.e., \( a_1 = a_2 = a_3 = t_{\text{MCR}}(j) \), then the absolute deadline of each job \( j_k \) is given by \( d_k \overset{\text{def}}{=} t_{\text{MCR}}(j) + D_i \). In Figure 2.6a, the deadline of each job is thus: \( d_1 = 15, d_2 = 16 \) and \( d_3 = 18 \) and according to EDF, this leads to the job priority assignment \( J_1 >_{\text{edf}} J_2 >_{\text{edf}} J_3 \) (and to a makespan of 12).

2. Starting from the previous release pattern in which all the jobs are released simultaneously at time \( t_{\text{MCR}}(j) \), one can slightly move backward the release time of job \( j_3 \) (for instance) in such a manner that \( j_3 \) is released at time \( t_{\text{MCR}}(j) - 5 \) (see Figure 2.6b). Its absolute deadline \( d_3 \) is thus shifted to time \( t_{\text{MCR}}(j) + 13 \) and since no assumption is made about the schedule before time \( t_{\text{MCR}}(j) \), we can suppose that \( j_3 \) did not execute before \( t_{\text{MCR}}(j) \). Therefore, the processing time of \( j_3 \) at time \( t_{\text{MCR}}(j) \) is \( C_3 = 5 \) and the job priority assignment resulting from this new release pattern is \( J_3 >_{\text{edf}} J_2 >_{\text{edf}} J_1 \) (leading to a makespan of 10).

In the particular case of EDF, shifting the absolute deadline of these three jobs by distinct amplitudes can modify their relative priority and a possibly large amount of job priority assignments can be derived for the same rem-job set \( \mathcal{J}_{\text{wc}} \). The lemma follows.

Since from Lemma 2.1 above, the prior knowledge of the worst-case rem-job set does not allow for determining a unique job priority assignment for FJP schedulers, we refine the notation of the upper-bounds on the idle-instants as follows: the upper-bounds on the idle-instants are denoted by \( \overline{\text{idle}}_k(j, \pi, \mathcal{P}) \) when \( \mathcal{P} \) is explicitly specified (in the context of FTP scheduler) and by \( \overline{\text{idle}}_k(j, \pi) \) otherwise, with the interpretation that for
(a) Assuming that the three jobs are released simultaneously upon the MCR\((j)\) allows to derive a first job priority assignment.

(b) Another job priority assignment can be derived by slightly modifying the release pattern of the jobs. Note that this modification leads to another makespan.

Figure 2.6: With FJP schedulers, multiple job priority assignments can be derived from the same worst-case rem-job set.

every job priority assignment \(X\):

\[
\overline{idle}_k(J, \pi) \geq \overline{idle}_k(J, \pi, X)
\]

Basically, the notation \(\overline{idle}_k(J, \pi, \mathcal{P})\) will be used when we will address FTP schedulers (since the job priority assignment is known beforehand for every mode transition) whereas we will use the notation \(\overline{idle}_k(J, \pi)\) for FJP schedulers since the job priority assignment is not known. It goes without saying that the prior knowledge of the jobs priority assignment allows for establishing tighter upper-bounds on the idle-instants, i.e., the upper-bounds \(\overline{idle}_k(J, \pi, \mathcal{P})\) are tighter than \(\overline{idle}_k(J, \pi)\). From these notations, it results that \(\overline{idle}_m(J, \pi)\) (and this also holds for \(\overline{idle}_m(J, \pi, \mathcal{P})\)) is an upper-bound on the
makespan and the condition in Validity Test 2.1 given on page 89 can be rewritten as

\[
\overline{\text{idlem}}(J_i^{wc}, \pi) \leq \min_{j \neq i} \left\{ \min_{1 \leq k \leq n_j} \{ D_k^j(M') \} \right\}
\]

for FJP schedulers and as

\[
\overline{\text{idlem}}(J_i^{wc}, \pi, P^i) \leq \min_{j \neq i} \left\{ \min_{1 \leq k \leq n_j} \{ D_k^j(M') \} \right\}
\]

for FTP schedulers, where \( P^i \) is the job priority assignment used by the old-mode scheduler \( S^i \). Mathematical expressions of these upper-bounds \( \overline{\text{idlem}}(J_i^{wc}, \pi) \) and \( \overline{\text{idlem}}(J_i^{wc}, \pi, P^i) \) on the \( m \)th idle-instants are defined for both identical and uniform platforms in Sections 2.6–2.9. The details of the validity algorithm for AM-MSO are given by Algorithm 4, where the upper-bounds \( \overline{\text{idlem}}(J_i^{wc}, \pi) \) must be replaced with

---

**Algorithm 4: Validity Test for AM-MSO**

**Input:** A multimode real-time application \( \tau = \{ \tau^1, \tau^2, \ldots, \tau^x \} \)

**Output:** A validity test for AM-MSO

```
begin
forall i, j ∈ [1, x] such that i ≠ j do
    τ_disabled ← τ^j;
    τ_enabled ← ∅;
    π_new ← ∅;
    Sort τ_disabled by non-decreasing order of mode change deadlines;
for (k = 1; k ≤ m; k++) do
    π_new ← π_new ∪ π_k;
forall (τ_j ∈ τ_disabled) do
    if (\( D_j^k(M') < \overline{\text{idlem}}(J_i^{wc}, \pi) \)) then return False;
    if (sched(\( π_{new}^k, S^j, τ_{enabled} \cup \{ τ_j^i \} \) == True) then
        τ_enabled ← τ_enabled ∪ \{ τ_j^i \};
        τ_disabled ← τ_disabled \{ τ_j^i \};
    end if
end for
end forall
return True;
end
```
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

\( \text{idle}_m(\mathcal{F}_i^{\text{wc}}, \pi, \mathcal{P}) \) at line 10 if the old-mode scheduler is FTP. Notice that this algorithm enables new-mode tasks only at the instants \( \text{idle}_k(\mathcal{F}_i^{\text{wc}}, \pi) \) (with \( k = 1, 2, \ldots, m \)). That is, it implicitly considers that every instant at which processors become available to the new-mode tasks are as late as possible. As a consequence, if all the mode change deadlines are met while running Algorithm 4 then all these deadlines will be met during every mode transition at run-time\(^1\). Nevertheless, the fact that Algorithm 4 simulates every idle-instant of every mode transition by its corresponding upper-bound \( \text{idle}_k(\mathcal{F}_i^{\text{wc}}, \pi) \) brings about the following situation: during the actual execution of the application, there could be some intervals of time (during any mode transition) during which the set of currently enabled new-mode tasks benefits from more processors than during the execution of Algorithm 4. We will thoroughly discuss this kind of situation later in this chapter and we will prove that these situations do not jeopardize the schedulability of the application.

2.5 Some preliminary results for determining validity tests

2.5.1 Introduction to the three required key results

We can now be more precise about the key results that are required to establish a validity test. Actually, three key results are required (previously, we did not mentioned the analysis of the worst-case rem-job set):

**Result 1.** First, it must be shown that disabling the old-mode tasks upon any MCR does not jeopardize the schedulability of the rem-jobs when they continue to be scheduled by the old-mode scheduler. That is, it must be guaranteed that the absolute deadline \( d_{i,a,b}^j \) of every rem-job \( \tau_{i,a,b}^j \) is met during any mode transition from every mode \( M_i \).

**Result 2.** Second, we must determine the worst-case rem-job set \( \mathcal{F}_i^{\text{wc}} \) for every mode \( M_i \). Indeed, for every mode transition from mode \( M_i \) to any other mode \( M_{i'} \), our validity algorithm (see Algorithm 4 above) determines the upper-bounds on the idle-instants by basing its computations on the corresponding worst-case rem-job sets \( \mathcal{F}_i^{\text{wc}} \) (at line 10). In all cases (i.e., identical or uniform platforms and FJP or FTP schedulers), we will demonstrate that the worst-case rem-job set \( \mathcal{F}_i^{\text{wc}} \) of every mode \( M_i \) is the one that contains one job \( J_{i,\ell} \) for each task \( \tau_{i,\ell} \) and such that every

---

\(^1\)Because Algorithm 4 considers every transition between every pair of modes of the application.
2.5 Some preliminary results for determining validity tests

job \( J_\ell \in J^\text{rec} \) has a processing time equals to \( C^j_\ell \), i.e., the WCET of the task \( \tau^j_\ell \).

**Result 3.** Finally, we must establish a mathematical expression that provides, for any given set \( J \) of jobs and platform \( \pi \):

1. an upper-bound \( \text{idle}_k(J, \pi) \) (\( 1 \leq k \leq m \)) on each idle-instant \( \text{idle}_k(J, \pi, X) \), for every job priority assignment \( X \). This concerns FJP schedulers.
2. an upper-bound \( \text{idle}_k(J, \pi, \mathcal{P}) \) (\( 1 \leq k \leq m \)) on each idle-instant \( \text{idle}_k(J, \pi, \mathcal{P}) \), for a specific job priority assignment \( \mathcal{P} \). This concerns FTP schedulers.

Note that the protocol SM-MSO requires only an upper-bound on the makespan, i.e., on the \( m^{\text{th}} \) idle-instant.

In the next section, we prove the first key result in Lemma 2.4 for any uniform platform and strongly work-conserving scheduler (and the result holds for any identical platform and weakly work-conserving scheduler). Then in Section 2.5.3, Corollary 2.1 provides the demonstration of the second key result. Finally, the upper-bounds \( \text{idle}_k(J, \pi) \) are determined \( \forall k \in [1, m] \) in Section 2.6.1 (for identical platforms and weakly work-conserving FJP schedulers) and in Section 2.8.2 (for uniform platforms and strongly work-conserving FJP schedulers) while the upper-bounds \( \text{idle}_k(J, \pi, \mathcal{P}) \) are determined in Section 2.7.1 (for identical platforms and weakly work-conserving FTP schedulers) and in Section 2.9.1 (for uniform platforms and strongly work-conserving FTP schedulers).

### 2.5.2 Demonstration of the first key result

This section aims to prove the first key result introduced above for any uniform platform and strongly work-conserving scheduler, as well as any identical platform and weakly work-conserving scheduler. That is, we show that keeping scheduling the rem-jobs by the old-mode scheduler during every mode transition guarantees that every rem-jobs meets its deadline. This result, essential to the validity of both protocols SM-MSO and AM-MSO, is based on the notion of predictability introduced below. The issue of predictability has been studied in Ha and Liu [17, 18, 19] in the multiprocessor context from the following perspective.
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

Definition 2.13 (Predictability, from Ha and Liu [18])

Let $A$ denote a scheduling algorithm, and let $J = \{J_1, J_2, \ldots, J_n\}$ be a set of $n$ jobs, where each job $J_i = (a_i, c_i, d_i)$ is characterized by an arrival time $a_i$, a computing requirement $c_i$ and an absolute deadline $d_i$. Let $r_i$ and $f_i$ denote the time at which job $J_i$ starts and completes its execution (respectively) when $J$ is scheduled by $A$. Now, consider any set $J' = \{J'_1, J'_2, \ldots, J'_n\}$ of $n$ jobs obtained from $J$ as follows. Job $J'_i$ has an arrival time $a_i$, an execution requirement $c'_i \leq c_i$ and a deadline $d_i$ (i.e., job $J'_i$ has the same arrival time and deadline as $J_i$, and an execution requirement no larger than $J_i$’s). Let $r'_i$ and $f'_i$ denote the time at which job $J'_i$ starts and completes its execution (respectively) when $J'$ is scheduled by $A$. Algorithm $A$ is said to be predictable if and only if for any set of jobs $J$ and for any such $J'$ obtained from $J$, it is the case that $r'_i \leq r_i$ and $f'_i \leq f_i \forall i$.

Informally, Definition 2.13 claims that an upper-bound on the starting time and on the completion time of each job can be determined by analyzing the situation under the assumption that each job executes for its WCET. The result from the works [17, 18, 19] that we will be using can be stated as follows.

Lemma 2.2 (From Ha and Liu [17, 18] and [19])

On identical multiprocessor platforms, any global, preemptive, FJP and work-conserving scheduler (including both weakly and strongly work-conserving schedulers) is predictable.

More recently, Liliana Cucu-Grosjean and Joël Goossens [12] have studied the predictability of strongly work-conserving schedulers upon uniform (and unrelated) platforms and one of their results can be stated as follows.

Lemma 2.3 (From L. Cucu-Grosjean and J. Goossens [12])

On uniform multiprocessor platforms, any global, preemptive, FJP and strongly work-conserving scheduler is predictable.

Based on Lemmas 2.2 and 2.3, we prove in the following lemma that disabling the old-mode tasks upon the mode change requests does not jeopardize the schedulability during mode transitions. This lemma has been drawn from [24] and extended to
uniform platforms. It provides the first key result presented above for both identical platforms/weakly work-conserving schedulers and uniform platforms/strongly work-conserving schedulers.

**Lemma 2.4**

Let \( M^i \) and \( M^j \) denote two distinct modes. If the application is running in mode \( M^i \) and a MCR(\( j \)) occurs at time \( t_{\text{MCR}(j)} \) then every rem-job meets its deadline during the transition phase while being scheduled by the old-mode scheduler \( S^i \) (this lemma holds for both protocols SM-MSO and AM-MSO, as well as for both identical platforms/weakly work-conserving schedulers and uniform platforms/strongly work-conserving schedulers).

**Proof**

From our hypotheses, we know that the set of tasks \( \tau^i \) of the mode \( M^i \) is schedulable by \( S^i \) upon \( \pi \). When the MCR(\( j \)) occurs at time \( t_{\text{MCR}(j)} \), the transition protocol disables every old-mode task, which is equivalent to set the processing time of all their future jobs to zero. Since \( S^i \) is predictable (from Lemma 2.2 or 2.3 depending on the scheduler), the deadline of every rem-job is still met.

### 2.5.3 Demonstration of the second key result

In Corollary 2.1 below, we prove the second key result introduced on page 100. That is, for any uniform platform and strongly work-conserving FTP (or FJP) scheduler, as well as any identical platform and weakly work-conserving FTP (or FJP) scheduler, we prove that the worst-case rem-job set \( J^{\text{wc}}_i \) of every transition from any mode \( M^i \) is the set of jobs that contains one job \( J^i_\ell \) for each task \( \tau^i_\ell \) and such that every job \( J^i_\ell \in J^{\text{wc}}_i \) has a processing time equals to \( C^i_\ell \). This corollary is based on the following Lemma 2.5.

**Lemma 2.5**

Let \( \pi \) be any multiprocessor uniform platforms (including identical platforms) and let \( J \) and \( J' \) be any fixed set of \( n \) synchronous jobs such that \( J = \{J_1, J_2, \ldots, J_n\} \) of processing times \( c_1, c_2, \ldots, c_n \) and \( J' = \{J'_1, J'_2, \ldots, J'_n\} \) of processing times \( c'_1, c'_2, \ldots, c'_n \).
(a) An example of schedule $S$ upon a 5-processors uniform platform. The idle-instants $\text{idle}_k(j, \pi, \mathcal{P})$ are denoted by $\text{idle}_k$ for sake of clarity.

(b) An example of schedule $S'$ upon the same 5-processors uniform platform. Also for sake of clarity, the idle-instants $\text{idle}_k(j, \pi, \mathcal{P})$ and $\text{idle}_k'(j', \pi, \mathcal{P})$ are denoted by $\text{idle}_k$ and $\text{idle}_k'$, respectively. In this figure, we have by contradiction $\text{idle}_3 < \text{idle}_3'$.

Figure 2.7: During the time interval $[\text{idle}_3(j, \pi, \mathcal{P}), \text{idle}_3'(j', \pi, \mathcal{P})]$ 3 jobs are running in $S'$ while only 2 jobs are running in $S$. 

104
2.5 Some preliminary results for determining validity tests

(a) An example of schedule $S$ upon a 5-processors identical platform. The idle-instants $\text{idle}_k(j, \pi, \mathcal{P})$ are denoted by $\text{id}le_0$ for sake of clarity.

(b) An example of schedule $S'$ upon the same 5-processors identical platform. Also for sake of clarity, the idle-instants $\text{idle}_k(j, \pi, \mathcal{P})$ and $\text{idle}_k'(j', \pi, \mathcal{P})$ are denoted by $\text{id}le_0$ and $\text{id}le_0'$, respectively. In this figure, we have by contradiction $\text{id}le_3 < \text{id}le_3'$.

**Figure 2.8:** During the time interval $[\text{id}le_3(j, \pi, \mathcal{P}), \text{id}le_3'(j', \pi, \mathcal{P})]$ 3 jobs are running in $S'$ while only 2 jobs are running in $S$. 
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

For any job priority assignment $\mathcal{P}$, if there exists a bijective function between $J$ and $J'$ such that every job $J'_r \in J'$ is mapped to exactly one job $J_r \in J$ and such that $c'_r \leq c_r$, then the $k$th idle-instant $\text{idle}_k(J, \pi, \mathcal{P})$ ($\forall k \in [1, m]$) in the schedule of $J$ upon $\pi$ is not lower than the $k$th idle-instant $\text{idle}_k(J', \pi, \mathcal{P})$ in the schedule of $J'$, i.e., it holds $\forall k \in [1, m]$ that

$$\text{idle}_k(J', \pi, \mathcal{P}) \leq \text{idle}_k(J, \pi, \mathcal{P})$$

Proof

The proof is a consequence of the predictability of work-conserving schedulers (including both weakly and strongly work-conserving schedulers). Let $S$ and $S'$ denote the schedule of $J$ and $J'$ upon $\pi$ with $\mathcal{P}$, respectively. We denote by $\text{comp}_r$ and $\text{comp}'_r$ the completion time of any job $J_r$ in $S$ and $J'_r$ in $S'$, respectively. It follows from the fact that $c'_r \leq c_r$ ($\forall r \in [1, n]$) and from the predictability of the considered schedulers (shown in Lemma 2.2 or 2.3 depending on the schedulers) that $\forall r \in [1, n]$: \[ \text{comp}'_r \leq \text{comp}_r \] (2.2)

The proof is made by contradiction. Suppose that there exists $\ell \in [1, m]$ such that

$$\text{idle}_\ell(J, \pi, \mathcal{P}) < \text{idle}_\ell(J', \pi, \mathcal{P})$$

An example of schedules $S$ and $S'$ is illustrated on Figures 2.7a and 2.7b (respectively) on a 5-processors uniform platform where $\text{idle}_3(J, \pi, \mathcal{P}) < \text{idle}_3(J', \pi, \mathcal{P})$. Since the platform is uniform in this example, the scheduler is strongly work-conserving and both schedules $S$ and $S'$ form a staircase. Similarly, an example of schedules $S$ and $S'$ is illustrated on Figures 2.8a and 2.8b (respectively) on a 5-processors identical platform where we also assumed $\text{idle}_3(J, \pi, \mathcal{P}) < \text{idle}_3(J', \pi, \mathcal{P})$. Since the platform is identical in this example, the scheduler is assumed to be weakly work-conserving. In both Figures 2.7 and 2.8, we voluntarily omitted the details about the processor speeds, the jobs characteristics, etc. since they are useless in the scope of these examples. Furthermore, note that in both examples no job is released after time 0. By definition of the idle-instants we know that in the schedule of any set $\mathcal{J}$ of jobs upon any uniform or identical multiprocessor platform, we have $\forall k \in [1, m]$:
2.5 Some preliminary results for determining validity tests

1. the idle-instant $\text{idle}_k(J, \pi, P)$ corresponds to the completion time of a job,
2. there is no waiting job at time $\text{idle}_k(J, \pi, P)$ and,
3. there are at most $(m - k)$ running jobs at time $\text{idle}_k(J, \pi, P)$. “At most” since there can exist some $r > k$ such that $\text{idle}_r(J, \pi, P) = \text{idle}_k(J, \pi, P)$.

Since every idle-instant corresponds to the completion of a job, this implies that within the time interval $[\text{idle}_k(J, \pi, P), \text{idle}_{k+1}(J, \pi, P)]$ there are at most $(m - k)$ running jobs in $S$ while there are at least $(m - (k + 1))$ running jobs in $S'$. Therefore, within $[\text{idle}_k(J, \pi, P), \text{idle}_{k+1}(J', \pi, P)]$, at least one job (say $J_r$) is already completed in $S$ while $J'_r$ is still running in $S'$, and the fact that $J'_r$ completes later in $S'$ than $J_r$ in $S$ leads to a direct contradiction of Inequality 2.2. As we can see in both examples of Figures 2.7 and 2.8, three jobs are running in $S'$ during the time interval $[\text{idle}_k(J, \pi, P), \text{idle}_{k+1}(J', \pi, P)]$ while only two jobs are running in $S$, thus meaning that there is one job completed in $S$ and still running in $S'$.

Corollary 2.1

For any multiprocessor uniform platforms $\pi$ and for any transition of the application from mode $M_i$ to mode $M_j$, let $J_{\text{any}}$ denote any set of rem-jobs issued from the old-mode tasks and let $J_{\text{wc}}$ be the set of rem-jobs that contains one job $J_{r_i}$ for each task $\tau_{i_r}$ and such that every job $J_{r_i} \in J_{\text{wc}}$ has a processing time equals to $C_{i_r}$. The $k$th idle-instants $\text{idle}_k(J_{\text{wc}}, \pi) (\forall k \in [1, m])$ in the schedule of $J_{\text{wc}}$ is never lower than the $k$th idle-instant $\text{idle}_k(J_{\text{any}}, \pi)$ in the schedule of $J_{\text{any}}$, i.e., it holds $\forall k \in [1, m]$ that

$$\text{idle}_k(J_{\text{any}}, \pi) \leq \text{idle}_k(J_{\text{wc}}, \pi)$$

Proof

The proof is a consequence of Lemma 2.5 above. Let $c_{r_i}^{\text{wc}}$ and $c_{r_i}^{\text{any}}$ denote the processing time of job $J_r$ in $J_{\text{wc}}$ and $J_{\text{any}}$, respectively. By definition, $J_{\text{wc}}$ contains one job $J_r$ of processing time $C_{i_r}$ for each task $\tau_{i_r} \in \tau^i$, i.e., it holds $\forall \tau_{i_r} \in \tau^i$ that

$$c_{r_i}^{\text{wc}} = C_{i_r}$$

and thus we know by definition of $J_{\text{any}}$ that $\forall J_r \in J_{\text{any}},$
In addition, we know that there could be some jobs \( J_\ell \in J_i^{\text{wc}} \) such that \( J_\ell \notin J_i^{\text{any}} \) (since \( J_i^{\text{any}} \) does not necessarily contain one job for each old-mode task). For each such job \( J_\ell \), we consider in the following that \( J_\ell \in J_i^{\text{any}} \) but \( c^{\text{any}}_\ell = 0 \). Hence, the number of jobs in both \( J_i^{\text{wc}} \) and \( J_i^{\text{any}} \) can be assumed to be the same (we denote this number by \( n \)) and we know that there exists a bijective function between \( J_i^{\text{wc}} \) and \( J_i^{\text{any}} \) such that every job \( J_\ell \in J_i^{\text{wc}} \) is mapped to by exactly one job \( J_\ell \in J_i^{\text{any}} \) and such that \( c^{\text{any}}_\ell \leq c^{\text{wc}}_\ell \). This bijection is illustrated in Figure 2.9, where the set \( J_i^{\text{wc}} \) contains 8 jobs. Four jobs (\( J_2, J_5, J_7 \) and \( J_8 \)) have been added to \( J_i^{\text{any}} \) with a zero processing time in order to get this bijection between the two sets. Thanks to this bijection, we know from Lemma 2.5 that \( \forall r \in [1, m] \) we have

\[
\text{idle}_r(J_i^{\text{any}}, \pi) \leq \text{idle}_r(J_i^{\text{wc}}, \pi)
\]

and the corollary follows.

\[ c^{\text{any}}_r \leq c^{\text{wc}}_r \]

By definition, we know for every mode transition from any mode \( M_i \) that each \( \text{idle}_k(J_i^{\text{wc}}, \pi) \), \( \forall k \in [1, m] \), is an upper-bound on the \( k^{\text{th}} \) idle-instant in the schedule of \( J_i^{\text{wc}} \) upon \( \pi \) (this also holds for each upper-bound \( \text{idle}_k(J_i^{\text{wc}}, \pi, P) \) if the job priority assignment \( P \) is known beforehand). Thanks to Corollary 2.1 above, we are now aware
2.5 Some preliminary results for determining validity tests

that each upper-bound \( \text{idle}_k(J^{\text{wc}}, \pi) \) (and \( \text{idle}_k(J^{\text{wc}}, \pi, P) \)) is also an upper-bound on the \( k \)th idle-instant in the schedule of any other set of rem-jobs issued from the old-mode tasks (i.e., the tasks of \( \tau^i \)). That is, for every mode transition from any mode \( M^i \) we have \( \forall k \in [1, m]: \)

\[
\text{idle}_k(J^{\text{wc}}, \pi) \geq \text{idle}_k(J^{\text{wc}}, \pi, P) \geq \text{idle}_k(J^{\text{any}}, \pi)
\]

and

\[
\text{idle}_k(J^{\text{wc}}, \pi, P) \geq \text{idle}_k(J^{\text{wc}}, \pi, \mathcal{P}) \geq \text{idle}_k(J^{\text{any}}, \pi, \mathcal{P})
\]

where \( J^{\text{any}} \) denotes any set of rem-jobs issued from the tasks of \( \tau^i \). As a result, the instants \( \text{idle}_k(J^{\text{wc}}, \pi) \) (and \( \text{idle}_k(J^{\text{wc}}, \pi, P) \)), with \( k = 1, 2, \ldots, m \), can be considered as the largest instants at which new-mode tasks are enabled during every transition from every mode \( M^i \) and thus, these instants can be used in our validity algorithm given by Algorithm 4 on page 99.

2.5.4 Presentation of some base results

We prove below an interesting property that will be used in Lemmas 2.20 (page 154) and 4.15 (page 359).

Lemma 2.6

Let \( J = \{J_1, J_2, \ldots, J_n\} \) be any set of \( n \) synchronous jobs of processing times \( c_1, c_2, \ldots, c_n \) ordered by decreasing job priority, i.e., \( J_1 > J_2 > \cdots > J_n \). Let \( \pi = [s_1, s_2, \ldots, s_m] \) be any \( m \)-processors uniform platform such that \( s_i \geq s_{i-1} \) \( \forall i \in [2, m] \). Suppose that \( J \) is scheduled upon \( \pi \) by a strongly work-conserving scheduler using the job priority assignment \( \mathcal{P} \) and let \( \text{ms}(J, \pi) \) denote the exact makespan on that schedule. If the processing time \( c_r \) of any job \( J_r \in J \) is increased to \( c_r' > c_r \) then it holds that

\[
\text{ms}(J', \pi) \leq \text{ms}(J, \pi) + \frac{c_r' - c_r}{s_m}
\]

where \( J' \) is the set of jobs \( J'_1, J'_2, \ldots, J'_n \) of processing time \( c'_1, c'_2, \ldots, c'_n \) such that \( c'_i = c_i \) \( \forall i \neq r \) and \( c_r' > c_r \) is the increased processing time of \( J_r \).

Proof

The proof is based on the result of Lemma 2.5 and is obtained by contradiction. Suppose that
\[
ms(J', \pi) > ms(J, \pi) + \frac{c'_r - c_r}{s_m}
\]

which can be rewritten as
\[
idle_m(J', \pi) > idle_m(J, \pi) + \frac{c'_r - c_r}{s_m} \tag{2.3}
\]

From Lemma 2.5, we know that \(\forall k \in [1, m]:\)
\[
idle_k(J, \pi, P) \leq idle_k(J', \pi, P) \tag{2.4}
\]

and by definition of the idle-instants it holds that
\[
\sum_{k=1}^{m} idle_k(J, \pi, P) \cdot s_k = \sum_{i=1}^{n} c_i \tag{2.5}
\]
\[
\text{and } \sum_{k=1}^{m} idle_k(J', \pi) \cdot s_k = \sum_{i=1}^{n} c'_i \tag{2.6}
\]

The proof is made by the following development.

\[
\begin{aligned}
\sum_{k=1}^{m} idle_k(J', \pi) \cdot s_k \\
= \sum_{k=1}^{m-1} idle_k(J', \pi) \cdot s_k + idle_m(J', \pi) \cdot s_m \\
\geq \sum_{k=1}^{m-1} idle_k(J, \pi, P) \cdot s_k + idle_m(J', \pi) \cdot s_m \quad \text{(from Inequality 2.4)} \\
> \sum_{k=1}^{m-1} idle_k(J, \pi, P) \cdot s_k + \left(idle_m(J, \pi) + \frac{c'_r - c_r}{s_m}\right) \cdot s_m \quad \text{(from Inequality 2.3)} \\
> \sum_{k=1}^{m} idle_k(J, \pi, P) \cdot s_k + (c'_r - c_r) \\
> \sum_{i=1}^{n} c_i + c'_r - c_r \quad \text{(from Equality 2.5)} \\
> \sum_{i=1}^{n} c'_i \quad \text{since } c'_i = c_i \forall i \neq r \text{ and } c'_r > c_r
\end{aligned}
\]

This directly leads to a contradiction of Equality 2.6 and the lemma follows.

As mentioned earlier, considering the worst-case rem-job set in the validity Algorithm 4 of AM-MSO (at line 10) can bring about the following situation: during the actual execution of the application, there could be some intervals of time (during any
2.5 Some preliminary results for determining validity tests

(a) Illustration of the schedule assumed by the execution of Algorithm 4. In this schedule, new-mode tasks are enabled at each instant $\text{idle}_k$, $1 \leq k \leq m$.

(b) Illustration of a possible schedule during a transition from mode $M_i$ to mode $M_j$ in the actual execution of the application. Here, new-mode tasks are enabled at each instant $\text{idle}_k$, $1 \leq k \leq m$, where $\text{idle}_k \leq \text{idle}_k$.

Figure 2.10: Within the time interval $[\text{idle}_3, \text{idle}_4]$, the tasks in $\tau(1) \cup \tau(2) \cup \tau(3) \cup \tau(4)$ benefit from 4 processors in the actual schedule (Figure 2.10b) while only 3 processors are available to these tasks in the schedule assumed by Algorithm 4 (Figure 2.10a).
mode transition) during which the set of currently enabled new-mode tasks benefits from more (and faster) processors than during the execution of Algorithm 4. This kind of situation can occur upon identical and uniform platforms and for both FJP and FTP schedulers. We depict an example of such a situation in Figure 2.10, where we consider a uniform platform \( \pi \) composed of 5 processors. In this figure, the system is transiting from mode \( M_i \) to mode \( M_j \). Other details such as the processor speeds, the characteristics of the jobs and the job priority assignment are useless in the scope of this example.

For sake of clarity, Figure 2.10 uses the notations \( \text{idle}_i \) and \( \overline{\text{idle}}_i \) instead of \( \overline{\text{idle}}(J_i^{\text{nc}}, \pi) \) and \( \overline{\text{idle}}(J_i^{\text{nc}}, \pi) \), respectively. Figure 2.10a depicts the schedule assumed by the execution of Algorithm 4 in which new-mode tasks are enabled at each instant \( \overline{\text{idle}}_i \) \( (k = 1, 2, \ldots, m) \) in Figure 2.10b is the same as the set of tasks enabled at each instant \( \text{idle}_i \) in Figure 2.10a. That is, if \( \tau(k) \) temporarily denotes the set of tasks enabled at time \( \overline{\text{idle}}_k \), \( \forall k \in [1, m] \), then we know for instance that the set of tasks enabled at time \( \text{idle}_1 \) in Figure 2.10b is the same as the task set \( \tau(1) \) enabled at time \( \text{idle}_1 \) in Figure 2.10a. Let us temporarily naming this property the “equivalence property” and suppose that at time \( \text{idle}_3 \) in Figure 2.10a some tasks are enabled (i.e., \( \tau(3) \neq \phi \)) and at time \( \text{idle}_4 \) no task is enabled, i.e., \( \tau(4) = \phi \). Thanks to the equivalence property, we know that the tasks enabled at time \( \text{idle}_3 \) in Figure 2.10b are the tasks of \( \tau(3) \) and those enabled at time \( \text{idle}_4 \) are the tasks of \( \tau(4) \). Since we assumed in Figure 2.10b that \( \text{idle}_3 = \text{idle}_4 \), it holds that the tasks enabled at time \( \text{idle}_3 \) are the tasks of \( \tau(3) \cup \tau(4) = \tau(3) \) (since \( \tau(4) = \phi \)). It follows that in the time interval \( [\overline{\text{idle}}_3, \overline{\text{idle}}_4] \), only 3 processors are available to the task set \( \tau(1) \cup \tau(2) \cup \tau(3) \) in Figure 2.10a while 4 processors are available to this task set in Figure 2.10b. Moreover, during this time interval, the additional processor \( \pi_4 \) available to \( \tau(1) \cup \tau(2) \cup \tau(3) \) in Figure 2.10b is faster (or of equal speed) than every processor in the subset of processors \( \{\pi_1, \pi_2, \pi_3\} \) available to \( \tau(1) \cup \tau(2) \cup \tau(3) \) in Figure 2.10a.
2.5 Some preliminary results for determining validity tests

In Lemma 2.7 below, we prove that this kind of situation does not jeopardize the schedulability of the application during its execution. Lemma 2.7 is proved while considering uniform platforms and strongly work-conserving schedulers but one can easily show that it also holds for identical platforms and weakly work-conserving schedulers.

**Lemma 2.7 (From P. Meumeu Yomsi, V. Nelis and J. Goossens [30])**

Any strongly work-conserving scheduler that is able to schedule a task set \( \tau \) upon a uniform platform \( \pi = [s_1, \ldots, s_m] \) is also able to schedule \( \tau \) upon any uniform platform \( \pi' \) such that (i) \( \pi' \supseteq \pi \) and (ii) \( \forall \pi_k \in \pi' \) and \( \pi_k \not\in \pi \) we have \( s_k \geq s_m \).

**Proof**

To obtain the proof, it is sufficient to show the Lemma for \( \pi' = [s_1, \ldots, s_m, s_{m+1}] \) where \( s_{m+1} \geq s_m \). The proof is made by contradiction. Suppose there exists a task set \( \tau \) that is schedulable by a strongly work-conserving scheduler \( S \) upon \( \pi \), but not upon \( \pi' \supseteq \pi \). Consider the schedule upon \( \pi' \) of a particular set \( J \) of jobs issued from \( \tau \) that leads to a deadline miss, and let \( J' \) be another set of jobs derived from \( J \) reducing the processing time of each job \( J_i \) by the amount of time \( J_i \) executes upon the sub-platform \( \pi' \setminus \pi \), i.e., upon \( \pi_{m+1} \). Since the scheduler is strongly work-conserving, the schedule of \( J \) by \( S \) upon the processors in common with \( \pi \) is the same as the one that would be produced by \( S \) for \( J' \) upon platform \( \pi \). Since a deadline is missed in the schedule of \( J \) upon \( \pi' \), then a deadline is missed also in the schedule of \( J' \) upon \( \pi \). But since the scheduler is predictable from Lemma 2.3, a deadline would be missed on \( \pi' \) even (a fortiori) with the more demanding jobs set \( J \), leading to a contradiction. The lemma follows.

2.5.5 Organization for the third key result

To conclude this section, Figure 2.11 depicts a chart of the various contributions presented in this chapter, as well as the relation between them. In the next section, Lemma 2.11 establishes a mathematical expression providing an upper-bound \( \overline{\text{idle}}_k(J, \pi) \) on each idle-instant idle\( \text{idle}_k(J, \pi, X) \) (for all \( X \)) while considering identical platforms and FJP schedulers. Then, we derive in Corollary 2.2 an upper-bound \( \overline{\text{ms}}_{\text{ident}}(J, \pi) \) on the makespan. In Section 2.7, we focus on the particular case of FTP schedulers and we establish in Corollary 2.3 an upper-bound \( \overline{\text{idle}}_k(J, \pi, \mathcal{P}) \) on each idle-instant idle\( \text{idle}_k(J, \pi, \mathcal{P}) \). Due to the fact that the job priority assignment \( \mathcal{P} \) is known beforehand for FTP schedulers, these upper-bounds are more accurate than those proposed for FJP schedulers.
Figure 2.11: Chart representation of the contributions presented in this chapter, where the labels (a), (b), (c) and (d) illustrate the relation between the contributions: (a) the upper-bound $\overline{ms}_{1}^{\text{unif}}(J, \pi)$ is more pessimistic than $\overline{ms}_{1}^{\text{ident}}(J, \pi)$, (b) the upper-bound $\overline{ms}_{2}^{\text{unif}}(J, \pi)$ is a generalization of $\overline{ms}_{2}^{\text{ident}}(J, \pi)$, (c) we conjecture that the upper-bound $\overline{ms}_{3}^{\text{unif}}(J, \pi)$ is more pessimistic than $\overline{ms}_{2}^{\text{ident}}(J, \pi)$ and (d) the computation of $\overline{ms}_{1}^{\text{unif}}(J, \pi, \mathcal{P})$ and $\overline{ms}_{1}^{\text{ident}}(J, \pi, \mathcal{P})$ are very different for FTP schedulers, especially because of the difference in the definitions of weakly and strongly work-conserving schedulers.

Again, we also derive an upper-bound on the makespan in Corollary 2.4. Then, we consider uniform platforms. Lemma 2.15 establishes an upper-bound $\overline{idle}_{k}(J, \pi)$ on each idle-instant idle$_{k}(J, \pi, \mathcal{X})$ (for all $\mathcal{X}$) while considering FJP schedulers. Then, we derive in Corollary 2.5 an upper-bound $\overline{ms}_{1}^{\text{unif}}(J, \pi, \mathcal{X})$ on the makespan but unfortunately, this upper-bound is not an extension to uniform platform of the upper-bound $\overline{ms}_{1}^{\text{ident}}(J, \pi)$ provided earlier by Corollary 2.2. Indeed, we will demonstrate in Lemma 2.30 (page 172) that the upper-bound $\overline{ms}_{1}^{\text{unif}}(J, \pi)$ is always more pessimistic than $\overline{ms}_{1}^{\text{ident}}(J, \pi)$ in the sense that $\overline{ms}_{1}^{\text{unif}}(J, \pi) \geq \overline{ms}_{1}^{\text{ident}}(J, \pi)$ for every set $J$ of jobs and every identical platforms $\pi$ (this is the relation (a) in Figure 2.11). In Lemmas 2.22 and 2.28, we propose two other upper-bounds on the maximum makespan for uniform platforms and FJP schedulers. The upper-bound $\overline{ms}_{2}^{\text{unif}}(J, \pi)$ presented in Lemma 2.22 can be considered as an extension to uniform platforms of the upper-bound $\overline{ms}_{2}^{\text{ident}}(J, \pi)$ provided by Corollary 2.2.
2.6 Validity test for identical platforms and FJP schedulers

This section is organized as follows.

1. Since only FJP schedulers are considered here, we start our study by determining in Section 2.6.1 an upper-bound \( \overline{idle}_k(J, \pi) \) on each idle-instant \( idle_k(J, \pi, X) \), for every job priority assignment \( X \).

2. Second, we establish a sufficient validity test for the protocol SM-MSO, assuming weakly work-conserving FJP schedulers and identical platforms. Concerning the protocol AM-MSO, the upper-bounds \( idle_k(J, \pi) \) that we determine in the next section can be used at line 10 of the validity algorithm given by Algorithm 4.

2.6.1 Determination of upper-bounds on the idle-instants

Throughout this section, the notation \( J \) refers to any set of \( n \) jobs. For sake of clarity, we will use the notations \( \overline{idle}_k \) instead of \( \overline{idle}_k(J, \pi) \) and similarly, the notation \( idle_k \) will be used to denote the exact value of the \( k \)th idle-instant. Before introducing the computation of these upper-bounds \( \overline{idle}_k, 1 \leq k \leq m \), let us introduce the following result.

\[ ms^8_{unif}(J, \pi) \text{ proposed in Lemma 2.28 is not an extension of } ms^8_{ident}(J, \pi) \text{ to uniform platforms and we conjecture that } ms^8_{unif}(J, \pi) \text{ is always more pessimistic than } ms^8_{ident}(J, \pi) \text{ (relation (c)), i.e., for every set } J \text{ of jobs and every identical platform } \pi: ms^8_{ident}(J, \pi) \leq ms^8_{unif}(J, \pi). \]

Then, we focus on FTP schedulers, still upon uniform platforms, and we establish in Corollary 2.8 a more precise upper-bound \( \overline{idle}_k(J, \pi, P) \) on each idle-instant \( idle_k(J, \pi, X) \) than those proposed for FJP schedulers, as well as a more precise upper-bound on the makespan (see Corollary 2.9). This upper-bound on the makespan can be considered as an extension to uniform platforms of the upper-bound established in Lemma 2.4 for identical platforms and FTP scheduler (this is the relation (d) in Figure 2.11). However, the computation of these two upper-bounds (in the identical and uniform cases) is very different because of the difference in the notions of weakly and strongly work conserving schedulers.
Lemma 2.8 (From V. Nelis, J. Goossens and B. Andersson [24])

Suppose that $J$ is ordered by non-decreasing job processing times, i.e., $c_1 \leq c_2 \leq \cdots \leq c_n$. Then, whatever the job priority assignment we have $\forall j, k \in [1, m]$ such that $j < k$:

$$\text{idle}_j \geq \text{idle}_k - c_{n-m+k}$$

Proof

The proof is made by contradiction. Suppose that there are $j$ and $k$ in $[1, m]$ such that $j < k$ and

$$\text{idle}_j < \text{idle}_k - c_{n-m+k}$$

By definition of the idle-instants, we know that $\text{idle}_k \leq \text{idle}_{k+1}$ $\forall k \in [1, m-1]$. Therefore, we know from the above inequality that the following $(m - k + 1)$ inequalities hold:

$$\text{idle}_j < \text{idle}_k - c_{n-m+k} \Rightarrow \text{idle}_j < \text{idle}_{k+1} - c_{n-m+k} \Rightarrow \text{idle}_j < \text{idle}_{k+2} - c_{n-m+k} \Rightarrow \cdots \Rightarrow \text{idle}_j < \text{idle}_m - c_{n-m+k}$$

These $(m - k + 1)$ inequalities suggest that at time $\text{idle}_j$ there remain exactly $(m - k + 1)$ jobs with a remaining processing time larger than $c_{n-m+k}$, each running on a different processor. However we know that there can only be at most $(m - k)$ such jobs since we assumed that $c_1 \leq c_2 \leq \cdots \leq c_n$. This therefore leads to a contradiction and the lemma follows.

Based on this result, the following Lemma 2.9 published in [24] determines an upper-bound $\overline{\text{idle}}_k$ on each idle-instant $\text{idle}_k$, $1 \leq k \leq m$. Nevertheless, while we were writing this thesis we successfully established another upper-bound $\overline{\text{idle}}_k$ (proved in Lemma 2.11) and Lemma 2.12 proves that this alternative upper-bound is always tighter than that proposed in Lemma 2.9. Finally, based on this new alternative upper-bound, Corollary 2.2 derives an upper-bound on the makespan.
Lemma 2.9 (From V. Nelis, J. Goossens and B. Andersson [24])

Suppose that $J$ is ordered by non-decreasing job processing times, i.e., $c_1 \leq c_2 \leq \cdots \leq c_n$. Then, whatever the job priority assignment, an upper-bound on the idle-instant $\text{idle}_k$, $1 \leq k \leq m$, is given by

$$\text{idle}_k \equiv \begin{cases} c_k & \text{if } n = m \\ \max_{i=0}^{n-m+k-1} \left\{ \frac{\sum_{j=1}^{n} c_j - \sum_{j=i+1}^{n-m+k+1} c_j}{m} + \frac{\sum_{j=i+1}^{n-m+k+1} c_j}{m-k+1} \right\} & \text{otherwise } (n > m). \end{cases} \quad (2.7)$$

Proof

The case where $n = m$ is obvious. Otherwise, we prove the lemma by contradiction. Suppose that there is a $k \in [1, m]$ such that $\text{idle}_k > \text{idle}_k$. The following properties hold:

(a) $\forall i > k$: $\text{idle}_i \geq \text{idle}_k$ (by definition of the idle-instants).

(b) $\forall i < k$: $\text{idle}_i \geq \text{idle}_k - c_{n-m+k}$ (from Lemma 2.8).

Obviously, we know that

$$\sum_{i=1}^{m} \text{idle}_i = \sum_{i=1}^{k-1} \text{idle}_i + \text{idle}_k + \sum_{i=k+1}^{m} \text{idle}_i$$

By applying both properties (a) and (b) to the right-hand side, we get

$$\sum_{i=1}^{m} \text{idle}_i \geq \sum_{i=1}^{k-1} (\text{idle}_k - c_{n-m+k}) + \text{idle}_k + \sum_{i=k+1}^{m} \text{idle}_k$$

$$\geq (k - 1) \cdot (\text{idle}_k - c_{n-m+k}) + \text{idle}_k + (m - k) \cdot \text{idle}_k$$

$$\geq m \cdot \text{idle}_k - (k - 1) \cdot c_{n-m+k}$$

Since by assumption $\text{idle}_k > \text{idle}_k$, replacing $\text{idle}_k$ with $\text{idle}_k$ in the above inequality yields

$$\sum_{i=1}^{m} \text{idle}_i > m \cdot \text{idle}_k - (k - 1) \cdot c_{n-m+k} \quad (2.8)$$

Now, let $\ell$ be a value of $i$ which maximizes $\text{idle}_k$ in Expression 2.7, i.e., $\ell$ is such that $0 \leq \ell \leq n - m + k - 1$ and $\forall r \in [0, n - m + k - 1]$: 

\[117\]
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

\[
\frac{\sum_{j=1}^{n} c_j - \sum_{j=\ell+1}^{\ell+m-k+1} c_j}{m} + \frac{\sum_{j=\ell+1}^{\ell+m-k+1} c_j}{m-k+1} \geq \frac{\sum_{j=1}^{n} c_j - \sum_{j=r+1}^{r+m-k+1} c_j}{m} + \frac{\sum_{j=r+1}^{r+m-k+1} c_j}{m-k+1}
\]  \quad (2.9)

Inequality 2.8 can be rewritten as

\[
\sum_{j=1}^{m} \text{idle}_{i} > m \cdot \left( \frac{\sum_{j=1}^{n} c_j - \sum_{j=\ell+1}^{\ell+m-k+1} c_j}{m} + \frac{\sum_{j=\ell+1}^{\ell+m-k+1} c_j}{m-k+1} \right) - (k-1) \cdot c_{n-m+k}
\]

and it yields

\[
\sum_{j=1}^{m} \text{idle}_{i} > \sum_{j=1}^{n} c_j - \sum_{j=\ell+1}^{\ell+m-k+1} c_j + \frac{m}{m-k+1} \cdot \sum_{j=\ell+1}^{\ell+m-k+1} c_j - (k-1) \cdot c_{n-m+k}
\]

\[
> \sum_{j=1}^{n} c_j + \left( \frac{m}{m-k+1} - 1 \right) \cdot \sum_{j=\ell+1}^{\ell+m-k+1} c_j - (k-1) \cdot c_{n-m+k}
\]

\[
> \sum_{j=1}^{n} c_j + \frac{k-1}{m-k+1} \cdot \sum_{j=\ell+1}^{\ell+m-k+1} c_j - (k-1) \cdot c_{n-m+k}
\]  \quad (2.10)

By definition of \( \ell \), we know from Expression 2.9 that \( \forall r \in [0, n-m+k-1] \) we have

\[
\frac{\sum_{j=1}^{n} c_j - \sum_{j=\ell+1}^{\ell+m-k+1} c_j}{m} + \frac{\sum_{j=\ell+1}^{\ell+m-k+1} c_j}{m-k+1} \geq \frac{\sum_{j=1}^{n} c_j - \sum_{j=r+1}^{r+m-k+1} c_j}{m} + \frac{\sum_{j=r+1}^{r+m-k+1} c_j}{m-k+1}
\]

and rewriting this yields

\[
\frac{\sum_{j=1}^{n} c_j}{m} - \frac{\sum_{j=\ell+1}^{\ell+m-k+1} c_j}{m} + \frac{\sum_{j=\ell+1}^{\ell+m-k+1} c_j}{m-k+1} \geq \frac{\sum_{j=1}^{n} c_j}{m} - \frac{\sum_{j=r+1}^{r+m-k+1} c_j}{m} + \frac{\sum_{j=r+1}^{r+m-k+1} c_j}{m-k+1}
\]

Since the two first terms of each side of this inequality are identical, we can remove them from the inequality and this leads to

\[
\frac{\sum_{j=\ell+1}^{\ell+m-k+1} c_j}{m-k+1} \geq \frac{\sum_{j=r+1}^{r+m-k+1} c_j}{m-k+1}
\]

By multiplying both sides by \((k-1)\) we get \( \forall r \in [0, n-m+k-1] \):

\[
\frac{m}{m-k+1} \cdot \sum_{j=\ell+1}^{\ell+m-k+1} c_j \geq \frac{(k-1)}{m-k+1} \cdot \sum_{j=r+1}^{r+m-k+1} c_j \\
\geq \frac{(k-1)}{m-k+1} \cdot (m-k+1) \cdot c_{r+1} \\
\geq (k-1) \cdot c_{r+1}
\]
Replacing $r$ with $n - m + k - 1$ in the above inequality yields

$$\frac{(k - 1)}{m - k + 1} \cdot \sum_{j=\ell+1}^{\ell+m-k+1} c_j \geq (k - 1) \cdot c_{n-m+k}$$

and from the above inequality, Inequality 2.10 can be rewritten as

$$\sum_{i=1}^{m} \text{idle}_i > \sum_{j=1}^{n} c_j + (k - 1) \cdot c_{n-m+k} - (k - 1) \cdot c_{n-m+k}$$

$$> \sum_{j=1}^{n} c_j$$

which leads to a contradiction since it holds from the definition of the idle-instants that $\sum_{j=1}^{m} \text{idle}_j = \sum_{i=1}^{n} c_i$. The lemma follows.

As briefly (and partially) mentioned earlier, the following three Lemmas 2.10, 2.11 and 2.12 complete our previous work [24] as follows.

- In Lemma 2.10 we show that the above expression of $\text{idle}_k$ (given by Expression 2.7) is always maximal for $i = n - m + k - 1$.

- In Lemma 2.11, we propose another upper-bound on each idle-instant $\text{idle}_k$.

- In Lemma 2.12 we show that this new upper-bound is never larger than that provided by Expression 2.7.

**Lemma 2.10**

If $n > m$, Expression 2.7 is maximal for $i = n - m + k - 1$.

**Proof**

We are aware that the length of the indexes used in the following notations makes the notations hard to visualize. Some figures representing these quantities would have been very helpful from the reader viewpoint, but this proof is purely based on algebra, rather than on visualization. Then we advise the reader to not try to visualize these expressions. The proof is made by contradiction. Let $i = n - m + k - 1$ and let $\ell$ be any integer in $[0, n - m + k - 2]$. Thus, it holds that $i > \ell$. Suppose by contradiction of the lemma that
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

\[ \frac{\sum_{j=1}^{n} c_j - \sum_{j=\ell+1}^{\ell+n-k+1} c_j}{m} + \frac{\sum_{j=\ell+1}^{\ell+n-k+1} c_j}{m - k + 1} > \frac{\sum_{j=1}^{n} c_j - \sum_{j=i+1}^{i+n-k+1} c_j}{m} + \frac{\sum_{j=i+1}^{i+n-k+1} c_j}{m - k + 1} \]

Since \( i = n - m + k - 1 \), the right-hand side of the above inequality can be rewritten as

\[ \frac{\sum_{j=1}^{n} c_j - \sum_{j=\ell+1}^{\ell+n-k+1} c_j}{m} + \frac{\sum_{j=\ell+1}^{\ell+n-k+1} c_j}{m - k + 1} > \frac{\sum_{j=1}^{n} c_j - \sum_{j=n-m+k}^{n} c_j}{m} + \frac{\sum_{j=n-m+k}^{n} c_j}{m - k + 1} \]

By multiplying both sides by \( m \cdot (m - k + 1) \), we get

\[ (m-k+1) \left( \frac{\sum_{j=1}^{n} c_j - \sum_{j=\ell+1}^{\ell+n-k+1} c_j}{m} + \frac{\sum_{j=\ell+1}^{\ell+n-k+1} c_j}{m - k + 1} \right) + m \cdot \sum_{j=\ell+1}^{\ell+n-k+1} c_j > (m-k+1) \left( \frac{\sum_{j=1}^{n} c_j - \sum_{j=n-m+k}^{n} c_j}{m} + \frac{\sum_{j=n-m+k}^{n} c_j}{m - k + 1} \right) + m \cdot \sum_{j=n-m+k}^{n} c_j \]

Now, we can remove \( (m-k+1) \cdot \sum_{j=1}^{n} c_j \) from both sides, leading to

\[ -(m-k+1) \cdot \sum_{j=\ell+1}^{\ell+n-k+1} c_j + m \cdot \sum_{j=\ell+1}^{\ell+n-k+1} c_j > -(m-k+1) \cdot \sum_{j=n-m+k}^{n} c_j + m \cdot \sum_{j=n-m+k}^{n} c_j \]

And thus,

\[ (k-1) \cdot \sum_{j=\ell+1}^{\ell+n-k+1} c_j > (k-1) \cdot \sum_{j=n-m+k}^{n} c_j \]

If \( k = 1 \) then we obviously get \( 0 > 0 \) and the lemma follows. Otherwise, if \( k > 1 \) then dividing both sides by \( (k-1) \) yields

\[ \sum_{j=\ell+1}^{\ell+n-k+1} c_j > \sum_{j=n-m+k}^{n} c_j \]

Since \( c_1 \leq c_2 \leq \cdots \leq c_n \), the left-hand side of the above inequality is maximum when \( \ell \) is maximum, i.e., when \( \ell = n - m + k - 2 \) since \( \ell \in [0, n - m + k - 2] \). This yields

\[ \sum_{j=n-m+k-1}^{n-1} c_j > \sum_{j=n-m+k}^{n} c_j \]
2.6 Validity test for identical platforms and FJP schedulers

and by subtracting $\sum_{j=n-m+k}^{n-1} c_j$ from both sides of this inequality we get

$$c_{n-m+k-1} > c_n$$

which leads to a contradiction since $c_1 \leq c_2 \leq \cdots \leq c_n$. The lemma follows. Notice that, by setting $i = n - m + k - 1$, Expression 2.7 becomes

$$\text{idle}_k \overset{\text{def}}{=} \begin{cases} c_k & \text{if } n = m \\ \frac{\sum_{j=1}^{n} c_j - \sum_{j=n-m+k}^{n} c_j}{m} + \frac{\sum_{j=n-m+k}^{n} c_j}{m-k+1} & \text{otherwise (} n > m \). \end{cases}$$

thus providing Expression 2.11.

Thanks to the above Lemma 2.10, Expression 2.7 can be rewritten as follows:

$$\text{idle}_k \overset{\text{def}}{=} \begin{cases} c_k & \text{if } n = m \\ \frac{\sum_{j=1}^{n} c_j - \sum_{j=n-m+k}^{n} c_j}{m} + \frac{\sum_{j=n-m+k}^{n} c_j}{m-k+1} & \text{otherwise (} n > m \). \end{cases}$$

(2.11)

Lemma 2.11

Suppose that $J$ is ordered by non-decreasing job processing times, i.e., $c_1 \leq c_2 \leq \cdots \leq c_n$. Then, whatever the job priority assignment, an upper-bound on the idle-instant $\text{idle}_k$, $1 \leq k \leq m$, is given by

$$\text{idle}_k \overset{\text{def}}{=} \begin{cases} c_k & \text{if } n = m \\ \frac{\sum_{i=1}^{n} c_i + (k-1) \cdot c_{n-m+k}}{m} & \text{otherwise (} n > m \). \end{cases}$$

(2.12)

Proof

The case where $n = m$ is obvious. Otherwise, we prove the lemma by contradiction. Suppose that there exists $k \in [1, m]$ such that $\text{idle}_k > \text{idle}_k$. The following properties hold:
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

Prop. (a) \( \forall j > k: \text{idle}_j \geq \text{idle}_k \) (by definition of the idle-instants).

Prop. (b) \( \forall j < k: \text{idle}_j \geq \text{idle}_k - c_{n-m+k} \) (from Lemma 2.8).

Obviously, it holds that:

\[
\sum_{j=1}^{m} \text{idle}_j = \sum_{j=1}^{k-1} \text{idle}_j + \text{idle}_k + \sum_{j=k+1}^{m} \text{idle}_j
\]

By applying both properties (a) and (b) to the right-hand side, we get

\[
\sum_{j=1}^{m} \text{idle}_j \geq \sum_{j=1}^{k-1} (\text{idle}_k - c_{n-m+k}) + \text{idle}_k + \sum_{j=k+1}^{m} \text{idle}_k
\]

\[
\geq (k-1)(\text{idle}_k - c_{n-m+k}) + \text{idle}_k + (m-k) \cdot \text{idle}_k
\]

leading to

\[
\sum_{j=1}^{m} \text{idle}_j \geq m \cdot \text{idle}_k - (k-1) \cdot c_{n-m+k}
\]

Since by hypothesis \( \text{idle}_k > \overline{\text{idle}}_k \), replacing \( \text{idle}_k \) with \( \overline{\text{idle}}_k \) in the above inequality leads to

\[
\sum_{j=1}^{m} \text{idle}_j > m \cdot \overline{\text{idle}}_k - (k-1) \cdot c_{n-m+k}
\]

\[
> m \left( \frac{\sum_{i=1}^{n} c_i + (k-1) \cdot c_{n-m+k}}{m} \right) - (k-1) \cdot c_{n-m+k}
\]

\[
> \sum_{i=1}^{n} c_i
\]

This leads to a contradiction since it holds from the definition of the idle-instants that

\[
\sum_{j=1}^{m} \text{idle}_j = \sum_{i=1}^{n} c_i
\]
2.6 Validity test for identical platforms and FJP schedulers

Lemma 2.12

The upper-bounds \( \overline{\text{idle}}_k \) (with \( k = 1, 2, \ldots, m \)) provided by Expression 2.12 are never larger than those provided by Expression 2.11.

Proof

The proof is made by contradiction. Let \( k \) be any integer in \([1, m]\). Let \( \overline{\text{idle}}_{\text{old}}^\text{new} \) denote the upper-bound provided by Expressions 2.11 and 2.12, respectively, and suppose that \( \overline{\text{idle}}_{\text{old}}^\text{new} > \overline{\text{idle}}_{\text{old}}^\text{old} \). We get

\[
\frac{\sum_{j=1}^{n} c_j + (k - 1) \cdot c_{n-m+k}}{m} > \frac{\sum_{j=1}^{n} c_j - \sum_{j=n-m+k}^{n} c_j}{m} + \frac{\sum_{j=n-m+k}^{n} c_j}{m - k + 1}
\]

By multiplying both sides by \( m \cdot (m - k + 1) \) we get

\[
(m - k + 1) \left( \sum_{j=1}^{n} c_j + (k - 1) \cdot c_{n-m+k} \right) > (m - k + 1) \left( \sum_{j=1}^{n} c_j - \sum_{j=n-m+k}^{n} c_j \right) + m \sum_{j=n-m+k}^{n} c_j
\]

and then, removing \( (m - k + 1) \cdot \sum_{j=1}^{n} c_j \) from both sides yields

\[
(m - k + 1) \cdot (k - 1) \cdot c_{n-m+k} > -(m - k + 1) \cdot \sum_{j=n-m+k}^{n} c_j + m \sum_{j=n-m+k}^{n} c_j
\]

Thus,

\[
(m - k + 1) \cdot (k - 1) \cdot c_{n-m+k} > (k - 1) \cdot \sum_{j=n-m+k}^{n} c_j
\]

If \( k = 1 \) then we obviously get \( 0 > 0 \) and the lemma follows. Otherwise, if \( k > 1 \) then dividing both sides by \( (k - 1) \) leads to

\[
(m - k + 1) \cdot c_{n-m+k} > \sum_{j=n-m+k}^{n} c_j
\]

In the right-hand side of the above inequality, there are \( m - k + 1 \) terms that are not lower than \( c_{n-m+k} \). This therefore leads to a contradiction since \( c_1 \leq c_2 \leq \cdots \leq c_n \).

The following corollary derives an upper-bound on the makespan from the upper-bound \( \overline{\text{idle}}_m \) provided by Expression 2.12.

123
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

Corollary 2.2

Suppose that \( J \) is ordered by non-decreasing job processing times, i.e., \( c_1 \leq c_2 \leq \cdots \leq c_n \). Then, whatever the job priority assignment, an upper-bound \( \overline{ms}^{\text{ident}}(J, \pi) \) on the makespan is given by

\[
\overline{ms}^{\text{ident}}(J, \pi) \overset{\text{def}}{=} \begin{cases} 
    c_n & \text{if } n = m \\
    \frac{\sum_{i=1}^{n-1} c_i}{m} + c_n & \text{otherwise}
\end{cases}
\]

Proof

Since the makespan corresponds to the \( m^{\text{th}} \) idle-instant, an upper-bound on the makespan is given by \( \overline{\text{idle}}_m \). Therefore, the proof is obtained by simply replacing \( k \) with \( m \) in Expression 2.12.

The accuracy of this upper-bound \( \overline{ms}^{\text{ident}}(J, \pi) \) is studied in Section 2.11 on page 177.

2.6.2 Determination of a validity test

From Corollary 2.2 and Corollary 2.1 (page 107), a sufficient validity test for the protocol SM-MSO upon identical platforms can be formalized as follows.

Validity Test 2.1 (For SM-MSO, FJP schedulers and identical platforms)

For any multimode real-time application \( \tau \) and any identical platform \( \pi \) composed of \( m \) processors, the protocol SM-MSO is valid provided that, for every mode \( M_i \),

\[
\overline{ms}^{\text{ident}}(J_i^{\text{wc}}, \pi) \leq \min_{j \neq i} \left\{ \min_{k=1}^{n_j} \{ D_k^j(M^i) \} \right\}
\]

where \( \overline{ms}^{\text{ident}}(J_i^{\text{wc}}, \pi) \) is defined as in Expression 2.13 and \( J_i^{\text{wc}} \) is defined as follows:

- \( J_i^{\text{wc}} \overset{\text{def}}{=} \{ J_{i1}, J_{i2}, \ldots J_{in} \} \)
- each job \( J_k \in J_i^{\text{wc}} \) has a processing time equal to the WCET \( C_k^i \) of task \( \tau_k^i \)
- \( J_i^{\text{wc}} \) is ordered by non-decreasing processing time.
Moreover, the upper-bounds \( \text{idle}_k(J, \pi) \) (with \( 1 \leq k \leq m \)) determined in Lemma 2.11 can be used at line 10 of the validity algorithm of AM-MSO (see Algorithm 4 page 99), as long as these upper-bounds are computed while assuming the worst-case rem-job set (i.e., \( J = J_{\text{wc}}^i \)) for the transitions from every mode \( M' \).

### 2.7 Validity test for identical platforms and FTP schedulers

This section is organized exactly as the previous one. That is,

1. since only FTP schedulers are considered we first determine in Section 2.7.1 an upper-bound \( \text{idle}_k(J, \pi, \mathcal{P}) \) on each idle-instant \( \text{idle}_k(J, \pi, \mathcal{P}) \), for any given job priority assignment \( \mathcal{P} \).

2. Then, we establish a sufficient validity test for the protocol SM-MSO for weakly work-conserving FTP schedulers and identical platforms.

#### 2.7.1 Determination of upper-bounds on the idle-instants

In this section, we focus on determining a mathematical expression for the upper-bounds \( \text{idle}_k(J, \pi, \mathcal{P}) \) where \( \pi \) still denotes any identical multiprocessor platform composed of \( m \) processors and \( \mathcal{P} \) is a specific given job priority assignment. Indeed, for a given FTP scheduler the priority of every task (and thus of every job) is known beforehand. This is why we focus here on determining an upper-bound \( \text{idle}_k(J, \pi, \mathcal{P}) \) (rather than \( \text{idle}_k(J, \pi) \) as in the previous section) on each idle-instant \( \text{idle}_k(J, \pi, \mathcal{P}) \)—because \( \mathcal{P} \) is now assumed to be known. This prior knowledge allows us to determine tighter upper-bounds than those proposed in the previous section. That is, the provided upper-bounds \( \text{idle}_k(J, \pi, \mathcal{P}) \) on the idle-instants are less generic than those for FJP schedulers. Once again, for sake of clarity, we will use the notations \( \text{idle}_k \) and \( \text{idle}_k \) instead of \( \text{idle}_k(J, \pi, \mathcal{P}) \) and \( \text{idle}_k(J, \pi, \mathcal{P}) \), respectively.

For any transition from a given mode \( M' \) to any other mode \( M'' \), the knowledge of the worst-case rem-job set \( J_{\text{wc}}^i \) and the fact that the job priority assignment is known beforehand allows to compute the exact maximum idle-instants \( \text{idle}_k \)—exact in the sense that they can actually be reached if every job executes for its WCET—simply by drawing the schedule of \( J_{\text{wc}}^i \) and by measuring the idle-instants \( \text{idle}_k \) in that schedule. Indeed, we know from Corollary 2.1 (on page 107) that each idle-instant \( \text{idle}_k(J_{\text{wc}}^i, \pi, \mathcal{P}) \) is an
upper-bound on the idle-instant $\text{idle}_k(J, \pi, P)$ derived from the schedule of any other set $J$ of rem-jobs. Before expressing mathematically these exact maximum idle-instants, let us introduce the following definition.

**Definition 2.14 (Processed work $\text{Work}^i_k$)**

Let $\pi$ denote any identical multiprocessor platform and let $S$ be any global, weakly work-conserving and FTP scheduler. Let $J = \{J_1, J_2, \ldots, J_n\}$ denote any set of $n$ jobs ordered by decreasing $S$-priority, i.e., $J_1 >_S J_2 >_S \cdots >_S J_n$ and let $S^i$ denote the schedule by $S$ of the $i$ highest priority jobs of $J$ upon $\pi$. The processed work $\text{Work}^i_k$ ($1 \leq k \leq m$ and $0 \leq i \leq n$) denotes the amount of processing time executed on processor $\pi_k$ in $S^i$.

![Figure 2.12: Illustration of the notion of processed work $\text{Work}^i_k$.](image)

In order to familiarize the reader with this notation $\text{Work}^i_k$, Figure 2.12 illustrates the schedule of 7 jobs $\{J_1, J_2, \ldots, J_7\}$ upon 4 identical processors, following the priority assignment: $J_1 > J_2 > \cdots > J_7$. In this schedule, we have for instance $\text{Work}^5_3 = 8$ because, in the schedule $S^5$ of the 5 highest priority jobs $J_1, J_2, J_3, J_4, J_5$, the amount of processing time units executed on $\pi_3$ is $c_2 + c_5 = 8$. Similarly, $\text{Work}^3_4 = 7$, $\text{Work}^3_3 = 2$, $\text{Work}^3_2 = 5$ and
2.7 Validity test for identical platforms and FTP schedulers

Work\textsuperscript{3} = 0 because, in the schedule S\textsuperscript{3} of jobs J\textsubscript{1}, J\textsubscript{2}, J\textsubscript{3}, we can see that 7 processing time units are executed on π\textsubscript{4} (i.e., the job J\textsubscript{1}), 2 processing time units are executed on π\textsubscript{3} (i.e., the job J\textsubscript{2}), 5 processing time units are executed on π\textsubscript{2} (i.e., the job J\textsubscript{3}) and no processing time unit is executed on π\textsubscript{1}. Notice that Work\textsubscript{0} = 0 ∀k = 1, 2, ..., m.

Lemma 2.13 provides the exact values of Work\textsubscript{i} (∀i = 1, 2, ..., n and ∀k = 1, 2, ..., m) when each job executes for its WCET. Then, Corollary 2.3 derives the exact maximum idle-instants idle\textsubscript{k} 1 ≤ k ≤ m, for the scheduling of any set J of n jobs upon any m-processors identical platform.

**Lemma 2.13**

Let π denote any identical multiprocessor platform composed of m processors. Let S be any global, weakly work-conserving and FTP scheduler and let J be any set of n jobs ordered by decreasing S-priority, i.e., J\textsubscript{1} >\textsubscript{S} J\textsubscript{2} >\textsubscript{S} \cdots >\textsubscript{S} J\textsubscript{n}. It holds ∀k ∈ [1, m] and ∀i ∈ [1, n] that

\begin{equation}
Work_{i}^{k} = \begin{cases} 
Work_{k-1}^{i} + c_{i} & \text{if } k = \max_{\ell \in [1, m]} \{ \arg\min_{\ell \in [1, m]} \{ Work_{\ell}^{i-1} \} \} \\
Work_{k-1}^{i-1} & \text{otherwise}
\end{cases}
\end{equation}

(2.14)

where Work\textsubscript{0} = 0 ∀k by definition of the processed work.

**Proof**

The proof directly follows from the definition of Work\textsubscript{i}∀i, k and from the second condition of our definition of a weakly work-conserving scheduler (see Definition 2.7, page 81). Indeed, whenever a subset P of several processors idle (or complete a job) at the same time, S dispatches the waiting job (if any) with the highest priority to the processor of P with the highest index (this is the reason for the condition “if k is the highest value of ℓ that minimizes Work\textsubscript{\ell}^{i-1}”).

**Corollary 2.3**

Let π be any identical multiprocessor platform composed of m processors. Let S be any global, weakly work-conserving and FTP scheduler and let J = {J\textsubscript{1}, J\textsubscript{2}, ..., J\textsubscript{n}} be any set of n jobs of respective processing time c\textsubscript{1}, c\textsubscript{2}, ..., c\textsubscript{n}. Suppose that J is ordered by decreasing S-priority, i.e., J\textsubscript{1} >\textsubscript{S} J\textsubscript{2} >\textsubscript{S} \cdots >\textsubscript{S} J\textsubscript{n}. 

127
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

If \( J \) is scheduled by \( S \) upon \( \pi \) then \( \text{idle}_k, 1 \leq k \leq m \), is the \( k \text{th} \) element of the vector \( \{\text{Work}_{k1}^\alpha, \text{Work}_{k2}^\alpha, \ldots, \text{Work}_{km}^\alpha\} \) ordered by non-decreasing order.

**Proof**

The proof directly follows from the definition of the processed work \( \text{Work}_{k}^\alpha \), \( \forall k \in [1, m] \).

**Corollary 2.4**

Let \( \pi \) be any identical multiprocessor platform composed of \( m \) processors. Let \( S \) be any global, weakly work-conserving and FTP scheduler and let \( J = \{J_1, J_2, \ldots, J_n\} \) be any set of \( n \) jobs of respective processing time \( c_1, c_2, \ldots, c_n \). Suppose that \( J \) is ordered by decreasing \( S \)-priority, i.e., \( J_1 >_S J_2 >_S \cdots >_S J_n \). If \( J \) is scheduled by \( S \) upon \( \pi \), then the maximum makespan \( m\text{ss}_{\text{ident}}(J, \pi) \) is given by \( \text{idle}_m \), where \( \text{idle}_m \) is determined as in Corollary 2.3.

### 2.7.2 Determination of a validity test

From Corollary 2.4, a sufficient validity test for the protocol SM-MSO can therefore be formalized as follows.

**Validity Test 2.2 (For SM-MSO, FTP schedulers and identical platforms)**

For any multimode real-time application \( \tau \) and any identical platform \( \pi \) composed of \( m \) processors, the protocol SM-MSO is valid provided that, for every mode \( M^i \),

\[
\overline{\text{ms}}_{\text{ident}}(\mathcal{J}^\text{wc}_i, \pi) \leq \min_{j \neq i} \left\{ \min_{k=1}^{n_j} \left\{ \mathcal{D}_k(M^i) \right\} \right\}
\]

where \( \overline{\text{ms}}_{\text{ident}}(\mathcal{J}^\text{wc}_i, \pi) \) is defined as in Corollary 2.4 and \( \mathcal{J}^\text{wc}_i \) is defined as follows:

\[
\mathcal{J}^\text{wc}_i \overset{\text{def}}{=} \{J_1, J_2, \ldots, J_n\}
\]

- each job \( J_k \in \mathcal{J}^\text{wc}_i \) has a processing time equal to the WCET \( C_k^i \) of task \( \tau_k^i \)
- \( \mathcal{J}^\text{wc}_i \) is ordered by decreasing \( S_i \)-priority.
Moreover, the upper-bounds $\overline{\text{idle}}_k(J, \pi, \mathcal{P})$ (where $1 \leq k \leq m$ and $\mathcal{P}$ corresponds to the job priority assignment of the old-mode scheduler $S'$) determined in Corollary 2.3 can be used at line 10 of the validity algorithm of AM-MSO (see Algorithm 4 page 99), as long as these upper-bounds are computed while assuming the worst-case rem-job set $\mathcal{J}_i^{wc}$ for the transitions from every mode $M'$.

2.8 Validity test for uniform platforms and FJP schedulers

2.8.1 Some useful observations

This section provides the reader with two important observations:

1. the maximum makespan determination problem is highly counter-intuitive upon uniform platforms, and

2. the methods for solving this problem cannot be straightforwardly extended from those proposed for identical multiprocessor platforms.

First, recall that the schedulers are now assumed to be strongly work-conserving since this section focuses on uniform platforms. For a given set of jobs, an intuitive idea for maximizing the makespan upon any $m$-processor uniform platform is to execute, at any time, the longest job upon the slowest processor, i.e., the shorter the computation requirement of a job, the higher its priority. We name this priority assignment “Shortest Job First” (SJF). However, we can show that this intuitive idea is erroneous, unfortunately. Indeed, as depicted in Figure 2.13a, the set $J$ of 4 jobs $J_1, J_2, J_3, J_4$ of respective processing time 4, 4, 16, 22, provides a makespan of 17.25 when scheduled upon the uniform platform $\pi = [1, 2]$ following SJF, i.e., $J_1 > J_2 > J_3 > J_4$, whereas the priority assignment $J_3 > J_1 > J_2 > J_4$ leads to a makespan of 19, as depicted in Figure 2.13b. Notice that the problem of determining in a polynomial time (i.e., without trying every priority assignment) a priority assignment leading to the maximum makespan remains an open question and is out of the scope of this thesis.

Another intuitive idea is to naively extend to uniform platforms the result (repeated below) of Corollary 2.2 on page 124. According to this Corollary 2.2, for any identical
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

(a) SJF leads to a makespan of 17.25. The number into brackets (next to the label of each processor) denotes the speed of the processor and each arrow represents the migration of a job from one processor to another one.

(b) The priority assignment $j_3 > j_1 > j_2 > j_4$ leads to a makespan of 19.

Figure 2.13: Counter-example proving that SJF policy does not always lead to the maximum makespan.
2.8 Validity test for uniform platforms and FJP schedulers

(a) The priority assignment $J_1 > J_2$ leads to a makespan of 4.

(b) The priority assignment $J_2 > J_1$ leads to a makespan of 3.5.

Figure 2.14: Upon uniform platforms, on the contrary to identical platforms, if the number of jobs is equal to the number of processors (i.e., $n = m$) then the makespan depends on the job priority assignment.

Platform $\pi$ composed of $m$ processors, an upper-bound on the makespan is given by

$$\bar{m_s}^{ident}(J, \pi) \overset{\text{def}}{=} \begin{cases} c_n & \text{if } (n = m) \\ \frac{\sum_{i=1}^{n-1} c_i}{m} + c_n & \text{otherwise} \end{cases}$$ (2.15)

where $c_i$ is assumed to be such that $c_i \geq c_{i-1}$ $\forall i \in [2, n]$. Upon identical platforms there is a sense in distinguishing the case $n = m$ from the case $n > m$, because the rem-jobs never migrate between processors during mode transitions. Therefore, in the particular case of $n = m$, the maximum makespan does not depend on the job priority assignment and can be determined exactly by $\bar{m_s}^{ident}(J, \pi) = c_n$. On the contrary, we can easily show that this property does not hold upon uniform platforms. That is, the maximum makespan in the case $n = m$ is not independent from the job priority assignment. This is shown through the following example depicted in Figure 2.14. Suppose the uniform platform $\pi = [1, 2]$ and the two jobs $J_1, J_2$ of processing time 4 and 6, respectively. If $J_1 > J_2$ (see Figure 2.14a) then $J_1$ completes on $\pi_2$ at time 2—time during which $J_2$ executes 2 execution units on $\pi_1$—and $J_2$ completes on $\pi_2$ at time 4, thus leading to a makespan of 4. On the other hand, if $J_2 > J_1$ (see Figure 2.14b) then $J_2$ completes on $\pi_2$ at time 3—time during which $J_1$ executes 3 execution units on $\pi_1$—and $J_1$ completes on $\pi_2$ at time 3.5, thus leading to a makespan of 3.5. As a result, the maximum makespan in the
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

case \( n = m \) depends on the job priority assignment on uniform platforms and the case \( n = m \) can no longer be distinguished from the case \( m < n \).

From this observation, naively extending Expression 2.15 to uniform platforms yields the following “1-piece” expression:

\[
\overline{m_{s_{0}}}^{\text{unif}}(J, \pi) \overset{\text{def}}{=} \frac{\sum_{i=1}^{n-1} c_i}{s(1)} + \frac{c_n}{s_m}
\]

(2.16)

(recall that \( s(1) = \sum_{i=1}^{n} s_i \)). Unfortunately, we show in Figure 2.15 that this extension does not provide an upper-bound on the maximum makespan. Upon the 3-processors platform \( \pi = [1, 2, 10] \), the set of jobs \( J = \{J_1, J_2, J_3\} \) of respective processing time 50, 80, 99 provides a maximum makespan of 20, reached using the job priority assignment \( J_1 > J_2 > J_3 \) (see Figure 2.15a). On the other hand, Expression 2.16 yields

\[
\overline{m_{s_{0}}}^{\text{unif}}(J, \pi) = \frac{50 + 80}{13} + \frac{99}{10} = 10 + 9.9 = 19.9 < 20
\]

This approximation made by Expression 2.16 is illustrated in Figure 2.15b. This simple counterexample is much more important than what it seems to be at first blush and we will deeply examine its impacts in Section 2.8.4 (page 138). In short, in addition to refute the fact that \( \overline{m_{s_{0}}}^{\text{unif}}(J, \pi) \) provides an upper-bound on the maximum makespan, it also refutes the main concept behind Expression 2.15. In this Expression 2.15, it can be easily shown that the term \( \sum_{i=1}^{n-1} c_i \) is an upper-bound on the dispatching time of \( J_n \). Therefore, this whole expression can be interpreted as

upper-bound on the makespan = upper-bound on the dispatching time of \( J_n + c_n \)

where \( J_n \) is the (or any) job with the largest processing time. In Expression 2.16, we can also easily show that the term \( \frac{\sum_{i=1}^{n-1} c_i}{s(1)} \) is an upper-bound on the dispatching time of \( J_n \) and at this time, \( J_n \) is dispatched to the fastest processor \( \pi_m \). But in this case (i.e., upon uniform platforms), on the contrary to identical platforms, adding the processing time \( c_n \) of job \( J_n \) to its “worst-case” dispatching time does not lead to an upper-bound on the maximum makespan. That is, the whole concept is erroneous on uniform platforms.
2.8 Validity test for uniform platforms and FJP schedulers

(a) This picture depicts a priority assignment leading to a makespan of 20. The speed of each processor is indicated into brackets next to its label. The numbers next to each job name $J_i$ is the amount of work processed by $J_i$ upon the allocated processor. For instance, job $J_1$ executes 50 execution units from time 0 to 5 on processor $\pi_3$, leading to its label $J_1(50)$.

(b) Approximation error made by Expression 2.16.

Figure 2.15: Illustration showing that Expression 2.15 cannot be naively extended to uniform platforms. Notice that we voluntarily exaggerated the error in this picture since the actual error is $20 - 19.9 = 0.1$.  

$$\frac{50 + 80}{1 + 2 + 10} = 10$$
We will deeply investigate this counterexample in Section 2.8.4 in order to figure out the underlying cause that makes this concept not extendable to uniform platforms. The conclusion of this section is that neither the “Shortest-Job-First” policy nor $mS^\text{unif}_0(J, \pi)$ lead to the maximum makespan.

### 2.8.2 Determination of upper-bounds on the idle-instants

Once more, but this time for any uniform platform $\pi$, this section focuses on determining a mathematical expression that provides an upper-bound $\text{idle}_k(J, \pi)$ on the $k$th idle-instant, $\forall k \in [1, m]$. For sake of clarity, the following two lemmas use the notations $\text{idle}_k$ instead of $\text{idle}_k(J, \pi)$ and similarly, the notation idle will be used to denote the exact value of the $k$th idle-instant. Also, we assume $m \leq n$ since the problem in the case $m > n$ reduces to the same problem upon the $n$ fastest processors (due to the second condition of the definition of strongly work-conserving schedulers). We start our analysis by determining in Lemma 2.14 a lower-bound $\text{idle}_k$ on each idle-instant $\text{idle}_k$, $1 \leq k \leq m$. Then, based on Lemma 2.14, Lemma 2.15 determines an upper-bound $\text{idle}_k$ on each idle-instant $\text{idle}_k$ and Corollary 2.5 derives an upper-bound on the maximum makespan (recall that the maximum makespan is simply given by $\text{idle}_m$).

**Lemma 2.14 (From P. Meumeu Yomsi, V. Nelis and J. Goossens [30])**

Let $\pi = [s_1, s_2, \ldots, s_m]$ be any $m$-processors uniform platform such that $s_i \geq s_{i-1}$ $\forall i$, $2 \leq i \leq m$. Let $J = \{J_1, J_2, \ldots, J_n\}$ be any set of $n$ jobs of respective processing times $c_1, c_2, \ldots, c_n$ such that $c_1 \leq c_2 \leq \cdots \leq c_n$. Let $S$ be the schedule of $J$ upon $\pi$ following any global, strongly work-conserving and FJP scheduler. A lower bound $\text{idle}_k$ on each idle-instant $\text{idle}_k$ ($1 \leq k \leq m$) in $S$ is given by

$$\text{idle}_k \overset{\text{def}}{=} \frac{\sum_{i=1}^{n-m+k} c_i}{s(1)} \quad (2.17)$$

**Proof**

According to the definition of the idle-instants, at most $(m - k)$ jobs are not completed at time $\text{idle}_k$, meaning that at least $(n - m + k)$ jobs are already completed. Let $J^{\text{any}}$ be any subset of $J$ composed of $r$ jobs, where $(n - m + k) \leq r \leq n$. Obviously, a lower bound $t$ on the instant at which the $r$ jobs of $J^{\text{any}}$ are completed is given by
2.8 Validity test for uniform platforms and FJP schedulers

\[ t \overset{\text{def}}{=} \sum_{j \in \text{any}} \frac{c_j}{s(1)} \]

and since \( c_1 \leq c_2 \leq \cdots \leq c_n \), \( t \) is minimal if

1. the number of jobs in \( \text{any} \) is low as possible, i.e., \( r = n - m + k \), and
2. the processing time of each job of \( \text{any} \) is low as possible.

As a result, \( t \) is minimum for \( \text{any} = \{J_1, J_2, \ldots, J_{n-m+k}\} \) and then yields a lower-bound for \( \text{idle}_k \). The lemma follows.

Lemma 2.15 (From P. Meumeu Yomsi, V. Nelis and J. Goossens [30])

Let \( \pi = [s_1, s_2, \ldots, s_m] \) be any \( m \)-processors uniform platform such that \( s_i \geq s_{i-1} \) \( \forall i \), \( 2 \leq i \leq m \). Let \( J = \{J_1, J_2, \ldots, J_n\} \) be any set of \( n \) jobs of respective processing times \( c_1, c_2, \ldots, c_n \) such that \( c_1 \leq c_2 \leq \cdots \leq c_n \). Let \( S \) be the schedule of \( J \) upon \( \pi \) following any global, strongly work-conserving and FJP scheduler. An upper-bound \( \text{idle}_k \) on each idle-instant \( \text{idle}_k \) (1 \( \leq \) \( k \) \( \leq \) \( m \)) in \( S \) is given by

\[ \text{idle}_k \overset{\text{def}}{=} \sum_{i=1}^{k} \frac{\text{idle}_j \cdot s_i}{s(k)} \]  

(2.18)

where \( s(k) \overset{\text{def}}{=} \sum_{i=k}^{m} s_i \) (as defined in Expression 2.1, page 78).

**Proof**

From the “staircase” property derived from the definition of a strongly work-conserving scheduler on uniform platform (see page 81 for details) and from the fact that all the jobs are assumed to be synchronous at time 0, we know that processor \( \pi_j \) becomes idle at time \( \text{idle}_j \), \( \forall j = 1, 2, \ldots, m \). Let \( w_j \) (1 \( \leq j \leq m \)) denotes the amount of work executed on processor \( \pi_j \) within \([0, \text{idle}_j] \), i.e., \( w_j \overset{\text{def}}{=} \text{idle}_j \cdot s_j \). The proof is made by contradiction. Let \( \ell \) be any integer in \([1, m]\) and suppose that \( \text{idle}_\ell > \text{idle}_\ell \). By definition of \( w_j \), we know that

\[ \sum_{j=1}^{m} w_j = \sum_{i=1}^{n} c_i \]  

(2.19)

and from the definition of \( w_j \) we know that
\[ \sum_{j=1}^{m} w_j = \sum_{j=1}^{m} \text{idle}_j \cdot s_j \]
\[ = \sum_{j=1}^{\ell-1} (\text{idle}_j \cdot s_j) + \sum_{j=\ell}^{m} (\text{idle}_j \cdot s_j) \]

By definition of the idle-instants, it holds \( \forall j \geq \ell \) that \( \text{idle}_j \geq \text{idle}_\ell \). Therefore, replacing “\( \text{idle}_j \)” with “\( \text{idle}_\ell \)” in the second term of the right-hand side of the above equality yields

\[ \sum_{j=1}^{m} w_j \geq \sum_{j=1}^{\ell-1} (\text{idle}_j \cdot s_j) + \sum_{j=\ell}^{m} (\text{idle}_\ell \cdot s_j) \]
\[ \geq \sum_{j=1}^{\ell-1} (\text{idle}_j \cdot s_j) + \text{idle}_\ell \cdot \sum_{j=\ell}^{m} s_j \]

By hypothesis we have \( \text{idle}_\ell > \overline{\text{idle}}_\ell \). Therefore, replacing \( \text{idle}_\ell \) with \( \overline{\text{idle}}_\ell \) in the right-hand side of the above inequality yields

\[ \sum_{j=1}^{m} w_j > \sum_{j=1}^{\ell-1} (\text{idle}_j \cdot s_j) + \overline{\text{idle}}_\ell \cdot \sum_{j=\ell}^{m} s_j \]
\[ > \sum_{j=1}^{\ell-1} (\text{idle}_j \cdot s_j) + \sum_{i=1}^{n} c_i - \sum_{i=1}^{\ell-1} \overline{\text{idle}}_i \cdot s_i \cdot \sum_{j=\ell}^{m} s_j \]
\[ > \sum_{j=1}^{\ell-1} (\text{idle}_j \cdot s_j) + \sum_{i=1}^{\ell-1} c_i - \sum_{i=1}^{\ell-1} \overline{\text{idle}}_i \cdot s_i \]
\[ > \sum_{i=1}^{n} c_i + \sum_{j=1}^{\ell-1} \left( (\text{idle}_j - \overline{\text{idle}}_j) \cdot s_j \right) \]

Since from Lemma 2.14 it holds that \( \overline{\text{idle}}_i \leq \text{idle}_i \) \( \forall i = 1, 2, \ldots, m \), it holds that

\[ \sum_{j=1}^{\ell-1} \left( (\text{idle}_j - \overline{\text{idle}}_j) \cdot s_j \right) \geq 0 \]
2.8 Validity test for uniform platforms and FJP schedulers

and thus

\[ \sum_{j=1}^{m} w_j > \sum_{i=1}^{n} c_i \]

leading to a contradiction with Equality 2.19. The theorem follows.

Corollary 2.5 (From P. Meumeu Yomsi, V. Nelis and J. Goossens [30])

Let \( \pi = [s_1, s_2, \ldots, s_m] \) be any \( m \)-processors uniform platform such that \( s_i \geq s_{i-1} \) \( \forall i \), \( 2 \leq i \leq m \). Let \( J = \{j_1, j_2, \ldots, j_n\} \) be any set of \( n \) jobs of respective processing times \( c_1, c_2, \ldots, c_n \) such that \( c_1 \leq c_2 \leq \cdots \leq c_n \). If \( S \) denotes the schedule of \( J \) upon \( \pi \) following any global, strongly work-conserving and FJP scheduler, then whatever the job priority assignment an upper-bound \( \overline{m_{s_1}}(J, \pi) \) on the makespan is given by

\[
\overline{m_{s_1}}(J, \pi) \overset{def}{=} \frac{1}{s_m} \left( \sum_{i=1}^{n} c_i - \sum_{i=1}^{m-1} \text{idle}_i \cdot s_i \right)
\]  

(2.20)

Proof

Since the makespan corresponds to the idle-instant \( \text{idle}_m \), an upper-bound on the makespan is given by \( \text{idle}_m \). Therefore, the proof is obtained by simply replacing \( k \) with \( m \) in Expression 2.18.

2.8.3 Determination of a validity test

From Corollary 2.5 and Lemma 2.1, a sufficient validity test for the protocol SM-MSO can therefore be formalized as follows.

Validity Test 2.3 (For SM-MSO, FJP schedulers and uniform platforms)

For any multimode real-time application \( \tau \) and any identical platform \( \pi \) composed of \( m \) processors, the protocol SM-MSO is valid provided that, for every mode \( M^i \),

\[
\overline{m_{s_1}}(\tau^i, \pi) \leq \min_{j \in \tau^i} \left\{ \min_{k=1}^{n_j} \left\{ D^j_k(M^i) \right\} \right\}
\]

where \( \overline{m_{s_1}}(\tau^i, \pi) \) is defined as \( \overline{m_{s_1}}(J, \pi) \) in Expression 2.20, considering that \( J \) is composed of \( n_i \) jobs of processing time \( C^i_1, C^i_2, \ldots, C^i_{n_i} \) such that \( C^i_j \geq C^i_{j-1} \) \( \forall j = 2, 3, \ldots, n_i \).
Similarly, the upper-bounds $\overline{\text{idle}}_k(J, \pi)$ (with $1 \leq k \leq m$) determined in Lemma 2.15 can be used at line 10 of the validity algorithm of AM-MSO (see Algorithm 4 page 99), as long as these upper-bounds are computed while assuming the worst-case rem-job set (i.e., $I = J_{i}^{wc}$) for the transitions from every mode $M_i$.

### 2.8.4 Another analysis of the maximum makespan: the main idea

The upper-bounds $\overline{\text{idle}}_k$ (for all $k = 1, 2, \ldots, m$) obtained from Expression 2.18 on page 135 are based only on the total processing time of all the jobs and on the lower-bounds $\text{idle}_k$. Consequently, this can ultimately lead to an unacceptably large value for $\text{idle}_m$, unfortunately. In the following we provide two other upper-bounds on the maximum makespan for FJP schedulers. These upper-bounds can be used in order to improve the Validity Test 2.3 for the protocol SM-MSO.

To understand the approach that we are about to follow, it is essential to deeply understand what was wrong with the “naive” Expression 2.16 (repeated below). Recall that in order to establish this expression, we naively extended the upper-bound $\overline{\text{ms}}_{\text{ident}}(J, \pi)$ (see Expression 2.13, page 124) to uniform platforms in a straightforward manner and we got:

$$\overline{\text{ms}}_{0}^{\text{unif}}(J, \pi) \defeq \frac{\sum_{i=1}^{n-1} c_i}{s_i} + \frac{c_n}{s_m} \quad (2.21)$$

This expression (as well as that of $\overline{\text{ms}}_{\text{ident}}(J, \pi)$) is based on the intuition that the maximum makespan is reached when the longest job is dispatched as late as possible and executes for its WCET. This intuition has revealed to be true for the case of identical platforms\(^1\), but not for the uniform case (as shown by the counterexample on page 133).

To understand why, let us focus on this counterexample. We considered a 3-processor platform $\pi = \{1, 2, 10\}$ and the set of jobs $I = \{J_1, J_2, J_3\}$ of respective processing time 50, 80 and 99. The maximum makespan is 20, reached by using the priority assignment $J_1 > J_2 > J_3$ (see the lower schedule in Figure 2.16), whereas we have seen that Expression 2.21 provides $\overline{\text{ms}}_{0}^{\text{unif}}(J, \pi) = 19.9 < 20$ (see the upper schedule in Figure 2.16). If $S_{\text{naive}}$ and $S_{\overline{\text{ms}}}$ denote the two schedules depicted in Figure 2.16 (issued respectively from the approximation $\overline{\text{ms}}_{0}^{\text{unif}}(J, \pi)$ and from the priority assignment $J_1 > J_2 > J_3$ leading to the maximum makespan), the reason why $\overline{\text{ms}}_{0}^{\text{unif}}(J, \pi)$ under-approximates the

---

\(^1\)Indeed, for identical platforms, it can easily be shown that the instant $\frac{\sum_{i=1}^{n-1} c_i}{s_m}$ is an upper-bound on the instant at which job $J_n$ can be dispatched to a processor.
Validity test for uniform platforms and FJP schedulers

Figure 2.16: Example of schedule in which the makespan is larger than that returned by \( \text{ms}_0^\text{unif}(J, \pi) \).

maximum makespan comes from the following fact:

If \( t \) denotes the instant at which job \( J_3 \) is dispatched to processor \( \pi_3 \) in \( S_{\text{ms}} \) (here, \( t = 12 \)), then during the time interval \([0, t] \), \( J_3 \) has executed a lower amount of execution units in the stairs of \( S_{\text{ms}} \) than upon \( \pi_3 \) in \( S_{\text{naive}} \).

In other words, in Figure 2.16, the cumulated green areas represent a lower amount of execution units than the red area. Indeed, \( J_3 \) executes \( 5 + 14 = 19 \) execution units within \([0, t] \) in \( S_{\text{ms}} \) whereas it executes 20 execution units on \( \pi_3 \) in \( S_{\text{naive}} \). As a result, the remaining processing time of \( J_3 \) at time \( t \) is higher in \( S_{\text{ms}} \) (here, 80) than in \( S_{\text{naive}} \) (here, 79), implying that \( J_3 \) completes later in \( S_{\text{ms}} \) than in \( S_{\text{naive}} \). This is the reason why the expression \( \text{ms}_0^\text{unif}(J, \pi) \) does not provide the maximum makespan in the example above: on uniform platforms, the schedule in which any job \( J \) reaches its maximum completion time...
is not necessarily the schedule in which $J_i$ is dispatched as late as possible. In the remainder of this section, we use the notations $\text{idle}_j(J_i, \pi, \mathcal{P})$ that are refinements of the idle-instants $\text{idle}_j(J, \pi, \mathcal{P})$ defined on page 94.

**Definition 2.15 (Idle-instants $\text{idle}_j(J_i, \pi, \mathcal{P})$)**

If $S_i$ denotes the schedule upon $\pi$ of only the jobs with a higher priority than $J_i$ according to $\mathcal{P}$, then $\text{idle}_j(J_i, \pi, \mathcal{P})$ is the earliest instant in $S_i$ at which at least $j$ processors idle.

Recall that, according to our definition of a strongly work-conserving scheduler upon uniform platforms (see Definition 2.8 on page 82) and especially because of its resulting “staircase” property, the subset of idle processors at each idle-instant $\text{idle}_j(J_i, \pi, \mathcal{P})$ ($\forall i, j$) contains at least the processors $\{\pi_1, \pi_2, \ldots, \pi_j\} \subseteq \pi$.

Obviously, determining an upper-bound on the makespan can be achieved by determining an upper-bound on the completion time of every rem-job. Following this idea, Lemma 2.16 below determines an upper-bound $\text{comp}_i(J, \pi, \mathcal{P})$ on the completion time of every job $J_i \in J$, assuming that the job priority assignment $\mathcal{P}$ is known beforehand. As explained in Section 2.4.2 (on page 93), this hypothesis is totally inconsistent while considering FJP schedulers, but it is temporary. Ultimately (before determining an upper-bound on the makespan), this hypothesis will be dropped.

**Lemma 2.16**

An upper-bound $\text{comp}_i(J, \pi, \mathcal{P})$ on the completion time of any job $J_i \in J$ is given by

$$\text{comp}_i(J, \pi, \mathcal{P}) \overset{\text{def}}{=} \text{idle}_m(J_i, \pi, \mathcal{P}) - \frac{\sum_{j=2}^{m} (\text{idle}_j(J_i, \pi, \mathcal{P}) - \text{idle}_{j-1}(J_i, \pi, \mathcal{P})) \cdot s_{j-1}}{s_m} + \frac{c_i}{s_m} \quad (2.22)$$

**Proof**

For sake of clarity, we will use in this proof the notations $\text{comp}_i$ and $\text{idle}_j^i$ instead of $\text{comp}_i(J, \pi, \mathcal{P})$ and $\text{idle}_j(J_i, \pi, \mathcal{P})$, respectively. This yields

$$\text{comp}_i \overset{\text{def}}{=} \text{idle}_m^i - \frac{\sum_{j=2}^{m} (\text{idle}_j^i - \text{idle}_{j-1}^i) \cdot s_{j-1}}{s_m} + \frac{c_i}{s_m}$$
The instant $t_i$ is determined in such a manner that the amount of execution units that can be executed into the red area is equivalent to the one that can be executed in the green areas.

Figure 2.17: Illustration of the time-instant $t_i$ (Figure 2.17a) and the early completion of job $J_i$ (Figure 2.17b).
**Figure 2.18:** Impossibility for job $J_i$ to complete later than $t_i + \frac{c_i}{s_m}$ while being executed in the green areas.

(b) In this illustration, we suppose by contradiction that $J_i$ completes later than instant $t_i + \frac{c_i}{s_m}$ while being executed in the green areas.
The proof directly follows from the construction of $\text{comp}_i$. Indeed, let us consider the schedule of every job of $J$ with a higher priority than $J_i$. This schedule is depicted in Figure 2.17a (page 141) where it forms a staircase framed by the thick black lines. In this figure, the green areas represent the allocation spaces on processors $\pi_1, \pi_2, \ldots, \pi_{m-1}$ in which $J_i$ can execute. The cumulated length of these green areas is given by

$$\sum_{j=2}^{m} (\text{idlem}^j_i - \text{idlem}^j_{j-1})$$

That is, this quantity represents the time available to the job $J_i$ in the time interval $[\text{idlem}^i_j, \text{idlem}^i_m]$. Therefore, if $J_i$ is scheduled in all these green areas then it executes $R_i \overset{\text{def}}{=} \sum_{j=2}^{m} (\text{idlem}^j_i - \text{idlem}^j_{j-1}) \cdot s_{j-1}$

execution units (assuming that it does not complete earlier than $\text{idlem}^i_m$). According to our interpretation of the processor speeds, the quantity $R_i / s_m$ is the time needed to execute $R_i$ execution units on processor $\pi_m$ and thus, the instant $t_i$ given by

$$t_i \overset{\text{def}}{=} \text{idlem}^i_m - \frac{R_i}{s_m} \quad (2.23)$$

is such that the amount of execution units executed on $\pi_m$ within $[t_i, \text{idlem}^i_m]$ (i.e., in the red area in Figure 2.17a) is equivalent to the one that can be executed on every processor $\pi_1, \pi_2, \ldots, \pi_{m-1}$ within $[\text{idlem}^i_j, \text{idlem}^i_m]$ (i.e., in all the green areas). In the remainder of this proof, we show that $\text{comp}_i \overset{\text{def}}{=} t_i + \frac{c_i}{s_m}$ (according to Expression 2.22) is an upper-bound on the completion time of $J_i$. We distinguish between 3 cases: $c_i > R_i$ (case 1), $c_i = R_i$ (case 2) and $c_i < R_i$ (case 3).

**Case 1: $c_i > R_i$.** If $J_i$ is scheduled in the green areas from time 0 to time $\text{idlem}^i_m$, the fact that $c_i > R_i$ implies that $J_i$ is not completed at time $\text{idlem}^i_m$, where its remaining execution time at that time is $c_i - R_i$. Therefore, $J_i$ completes on processor $\pi_m$ at time $\text{idlem}^i_m + \frac{c_i - R_i}{s_m}$, where

$$\text{idlem}^i_m + \frac{c_i - R_i}{s_m} = \text{idlem}^i_m - \frac{R_i}{s_m} + \frac{c_i}{s_m}$$

$$= t_i + \frac{c_i}{s_m} \quad \text{(from Expression 2.23)}$$
In this case, the instant $\text{comp}_i$ is therefore the exact completion time of $J_i$.

**Case 2:** $c_i = R_i$. If $J_i$ is scheduled in the green areas from time 0 to time $\text{idle}_m^i$, the fact that $c_i = R_i$ implies that $J_i$ completes exactly at time $\text{idle}_m^i$ and the following equalities obviously hold:

$$
\text{idle}_m^i = \text{idle}_m^i - \frac{c_i}{s_m} \quad \text{since } c_i = R_i
$$

$$
= t_i + \frac{c_i}{s_m}
$$

As for the previous case, $\text{comp}_i$ corresponds here to the exact completion time of $J_i$.

**Case 3:** $c_i < R_i$. In this case, job $J_i$ completes strictly before time $\text{idle}_m^i$ while being scheduled on processors $\pi_k$ with $k < m$ (i.e., in the green areas). That is, $J_i$ does not execute on processor $\pi_m$. This situation is illustrated in Figure 2.17b, where the execution of $J_i$ is represented in mauve. In this case, we prove that $\text{comp}_i$ is an upper-bound on the completion time of $J_i$ by contradiction. Consider Figure 2.18a where the red and green areas are depicted side by side. Let $\text{comp}_i$ denotes the completion time of job $J_i$ when executed in the green areas and suppose by contradiction that $\text{comp}_i > t_i + \frac{c_i}{s_m}$ (see Figure 2.18b). Thereby, it holds that

$$
\text{idle}_m^i - \text{comp}_i < \text{idle}_m^i - \left( t_i + \frac{c_i}{s_m} \right)
$$

In Figure 2.18b, let $w_{\text{red}}$ denote the amount of execution units executed in the red area within $\left[ t_i + \frac{c_i}{s_m}, \text{idle}_m^i \right]$, i.e.,

$$
w_{\text{red}} \overset{\text{def}}{=} \left( \text{idle}_m^i - \left( t_i + \frac{c_i}{s_m} \right) \right) \cdot s_m
$$

Then, let $w_{\text{green}}$ denote the remaining amount of execution units that can be executed in the green areas after the execution of $J_i$, i.e., within $\left[ \text{comp}_i, \text{idle}_m^i \right]$. It holds that

$$
w_{\text{green}} \leq \left( \text{idle}_m^i - \text{comp}_i \right) \cdot s_{m-1}
$$

$$
< \left( \text{idle}_m^i - \left( t_i + \frac{c_i}{s_m} \right) \right) \cdot s_{m-1} \quad \text{(from Inequality 2.24)}
$$
and because $s_{m-1} \leq s_m$, it holds that

$$w_{\text{green}} < \left( \text{idle}^i_m - \left( t_i + \frac{c_i}{s_m} \right) \right) \cdot s_m$$

$$< w_{\text{red}} \quad \text{(from Expression 2.25)}$$

Thus, adding $c_i$ to both sides of the above inequality leads to

$$w_{\text{green}} + c_i < w_{\text{red}} + c_i$$

and since we know that $w_{\text{green}} + c_i = R_i$ (see Figure 2.18b) we get

$$R_i < w_{\text{red}} + c_i$$

This means that the amount of execution units executed into the whole red area in Figure 2.17a is strictly greater than that executed into all the green areas. This leads to a contradiction since we know that instant $t_i$ is determined in such a manner that these two amounts of execution units are equivalent.

From the three cases presented above, we can conclude that the instant $\overline{\text{comp}}_i \overset{\text{def}}{=} t_i + \frac{c_i}{s_m}$ is an upper-bound on the completion time of $J_i$ and the lemma follows.

Based on Lemma 2.16 above, the following two sections respectively establish a second and third upper-bound on the makespan denoted by $\overline{\text{ms}}_{\text{unif}}^2(J, \pi)$ and $\overline{\text{ms}}_{\text{unif}}^3(J, \pi)$, respectively. In both sections, we follow exactly the same reasoning based on expression 2.22, i.e., the expression

$$\overline{\text{comp}}_i(J, \pi, \mathcal{P}) \overset{\text{def}}{=} \text{idle}_m(i, \pi, \mathcal{P}) - \sum_{j=2}^{m} \left( \text{idle}_j(i, \pi, \mathcal{P}) - \text{idle}_{j-1}(i, \pi, \mathcal{P}) \right) \cdot \frac{s_{j-1}}{s_m} + \frac{c_i}{s_m}$$

### 2.8.5 A second upper-bound on the makespan

The reasoning that we follow to derive the upper-bound $\overline{\text{ms}}_{\text{unif}}^2(J, \pi)$ on the makespan is composed of 7 consecutive steps described in details below.

**Step 1.** The fact that $\mathcal{P}$ is assumed to be known in the expression of $\overline{\text{comp}}_i(J, \pi, \mathcal{P})$ (see Expression 2.22) is problematic. Indeed, we focus here on FJP schedulers and for the
reasons explained on page 96, we cannot assume a specific job priority assignment for such schedulers. That is, ultimately, we need to determine an upper-bound \( \text{comp}_i(J, \pi, P(x)) \) such that for every job priority assignment \( X \),

\[
\text{comp}_i(J, \pi) \geq \text{comp}_i(J, \pi, X)
\]

In order to do so, we start by determining a lower-bound on the term

\[
\sum_{j=2}^{m} \left( \text{idle}_j(J, \pi, P) - \text{idle}_{j-1}(J, \pi, P) \right) \cdot s_{j-1}
\]

That is, we determine a lower-bound on the cumulated amount of execution units that can be executed in the green areas of Figure 2.17a. This lower-bound is determined in the following Lemma 2.17.

**Step 2.** The second step consists in replacing the term

\[
\sum_{j=2}^{m} \left( \text{idle}_j(J, \pi, P) - \text{idle}_{j-1}(J, \pi, P) \right) \cdot s_{j-1}
\]

with the lower-bound determined in Step 1 in the expression of \( \text{comp}_i(J, \pi, P) \). Recall that according to Expression 2.22, we have

\[
\text{comp}_i(J, \pi, P) \overset{\text{def}}{=} \text{idle}_m(J, \pi, P) - \frac{\sum_{j=2}^{m} \left( \text{idle}_j(J, \pi, P) - \text{idle}_{j-1}(J, \pi, P) \right) \cdot s_{j-1}}{s_m} + \frac{c_i}{s_m}
\]

and this replacement therefore leads to another upper-bound \( \text{comp}_i^{(2)}(J, \pi, P) \geq \text{comp}_i(J, \pi, P) \), where \( \text{comp}_i^{(2)}(J, \pi, P) \) is given by

\[
\text{comp}_i^{(2)}(J, \pi, P) \overset{\text{def}}{=} \left( 1 - \frac{s_1}{s_m} \right) \cdot \text{idle}_m(J, \pi, P) + \frac{1}{s_m} \cdot \left( c_i + s_1 \cdot \frac{\sum_{j=1}^{i-1} c_j}{s(1)} \right)
\]

In the following Lemma 2.18, we prove that \( \text{comp}_i^{(2)}(J, \pi, P) \) defined as above is indeed an upper-bound on the completion time of any job \( J_i \). This new upper-bound \( \text{comp}_i^{(2)}(J, \pi, P) \) is independent of the idle-instants \( \text{idle}_j(J, \pi, P) \), \( \forall j = 1, 2, \ldots, m-1 \), on the contrary to the expression of \( \text{comp}_i(J, \pi, P) \). That is, only the idle-instant \( \text{idle}_m(J, \pi, P) \) is still problematic in the expression of \( \text{comp}_i^{(2)}(J, \pi, P) \).
2.8 Validity test for uniform platforms and FJP schedulers

Step 3. The third step consists in showing for any set \( J \) of \( n \) jobs, any uniform platform \( \pi \) and any job priority assignment \( \mathcal{P} \) that, if there exists \( \ell \in [1, n] \) such that

\[
J_\ell \text{ is not the lowest priority job according to } \mathcal{P}
\]

and

\[
\text{comp}_{\ell}^{(2)}(J, \pi, \mathcal{P}) = \max_{i=1}^{n} \{ \text{comp}_{i}^{(2)}(J, \pi, \mathcal{P}) \}
\]

then there exists a job priority assignment \( \mathcal{P}' \) such that

\[
J_\ell \text{ is the lowest priority job according to } \mathcal{P}'
\]

and

\[
\text{comp}_{\ell}^{(2)}(J, \pi, \mathcal{P}') = \max_{i=1}^{n} \{ \text{comp}_{i}^{(2)}(J, \pi, \mathcal{P}') \}
\]

and

\[
\text{comp}_{\ell}^{(2)}(J, \pi, \mathcal{P}') \geq \text{comp}_{\ell}^{(2)}(J, \pi, \mathcal{P})
\]

This property is shown in the following Lemma 2.19.

Step 4. It follows from the result of the previous step that there always exists at least one job priority assignment (say \( \mathcal{P} \)) such that

\[
J_{\text{low}} \text{ is the lowest priority job according to } \mathcal{P}
\]

and

\[
\forall X: \quad \text{comp}_{\text{low}}^{(2)}(J, \pi, \mathcal{P}) \geq \max_{i=1}^{n} \{ \text{comp}_{i}^{(2)}(J, \pi, X) \}
\]

That is,

\[
\text{comp}_{\text{low}}^{(2)}(J, \pi, \mathcal{P}) \geq \text{maximum makespan}
\]

This consequence is shown in the following Corollary 2.6.

Step 5. Thanks to the result of the previous step, we know that we can bound the maximum makespan from above by focusing only on the completion time of the lowest priority job, considering every job priority assignment. That is, the maximum makespan is bounded from above by

\[
\max_{\forall X} \{ \text{comp}_{\text{low}}^{(2)}(J, \pi, X) \}
\]

(2.27)

where \( J_{\text{low}} \) denotes the lowest priority job in every job priority assignment \( X \). However, the job priority assignment \( \mathcal{P} \) that maximizes Expression 2.27 could be not unique, i.e., there can be more than one job priority assignment \( \mathcal{P} \) such that \( \text{comp}_{\text{low}}^{(2)}(J, \pi, \mathcal{P}) \geq \text{comp}_{\text{low}}^{(2)}(J, \pi, X) \) \( \forall X \). We denote the number of such job priority assignment by \( r \) and these \( r \) job priority assignment are denoted by \( \mathcal{P}_{i}^{\text{max}} \) with \( i = 1, 2, \ldots, r \). From the
definition of $P_i^{\text{max}}$ given by Expression 2.27, it therefore holds $\forall i, j \in [1, r]$ that

$$\overline{\text{comp}}_\text{low}^{(2)}(J, \pi, P_i^{\text{max}}) = \overline{\text{comp}}_\text{low}^{(2)}(J, \pi, P_j^{\text{max}})$$

Informally, all the job priority assignments $P_i^{\text{max}}$ (for $i = 1, \ldots, r$) have a value of $\overline{\text{comp}}_\text{low}^{(2)}(J, \pi, P_i^{\text{max}})$ that bounds the maximum makespan from above. In this fifth step, we improve this result in the following Lemma 2.20 by showing that among all these job priority assignments $P_i^{\text{max}}$, at least one of them assigns the lowest priority to the (or any) job with the largest processing time, i.e., there exists at least one $P_i^{\text{max}}$ (with $i \in [1, r]$) such that

- $J_{\text{low}}$ is the lowest priority job according to $P_i^{\text{max}}$
- and $c_{\text{low}} = \max_{j=1}^n \{c_j\}$

**Step 6.** The result of Step 5 can be reformulated as follows: there always exists at least one job priority assignment $P$ such that:

- $J_{\text{low}}$ is the lowest priority job according to $P$
- and $c_{\text{low}} = \max_{j=1}^n \{c_j\}$
- and $\forall X: \overline{\text{comp}}_\text{low}^{(2)}(J, \pi, P_i^{\text{max}}) \geq \max_{j=1}^n \{\text{comp}_\text{low}^{(2)}(J, \pi, X)\}$

i.e.,

$$\overline{\text{comp}}_\text{low}^{(2)}(J, \pi, P) \geq \text{maximum makespan}$$

In this sixth step, we determine a recursive expression providing an upper-bound on the maximum value of $\overline{\text{comp}}_\text{low}^{(2)}(J, \pi, X) \forall X$. That is, we provide an upper-bound $\overline{\text{comp}}_\text{low}^{(2)}(J, \pi)$ such that for every job priority assignment $X$:

$$\overline{\text{comp}}_\text{low}^{(2)}(J, \pi) \geq \overline{\text{comp}}_\text{low}^{(2)}(J, \pi, X)$$

This new upper-bound $\overline{\text{comp}}_\text{low}^{(2)}(J, \pi)$ is thus also an upper-bound on the maximum makespan and we denote it by $\overline{\text{ms}}_\text{unif}^{(2)}(J, \pi)$ instead of $\overline{\text{comp}}_\text{low}^{(2)}(J, \pi)$. This upper-bound $\overline{\text{ms}}_\text{unif}^{(2)}(J, \pi)$ is given in the following Lemma 2.21 and is obtained by a recursive expression based on SJF (Shortest Job First) policy. However, this result must be interpreted very carefully. Indeed, we show in this sixth step that the maximum value of $\overline{\text{comp}}_\text{low}^{(2)}(J, \pi, X) \forall X$ can be bounded from above by using SJF. But this result stems only
2.8 Validity test for uniform platforms and FJP schedulers

from the way in which \( \text{comp}^{(2)}_{\text{low}}(J, \pi, \mathcal{P}) \) has been written in Step 2 (Expression 2.26). The reader should keep in mind that the exact makespan is not necessarily maximum by using SJF as we presented a counterexample in Section 2.8.1 on page 129.

Step 7. Finally, we provide a non-recursive expression of \( \text{ms}^{\text{unif}}_2(J, \pi) \) in the following Lemma 2.22.

As mentioned in Step 1, the determination of our second upper-bound \( \text{ms}^{\text{unif}}_2(J, \pi) \) on the makespan starts with the lower-bound presented below. For sake of clarity, we use (once again) in the following two proofs the notations idle, \( \text{comp} \), and \( \text{comp}^{(2)} \) instead of idle(\( J, \pi, \mathcal{P} \)), \( \text{comp}(J, \pi, \mathcal{P}) \) and \( \text{comp}^{(2)}(J, \pi, \mathcal{P}) \), respectively.

Lemma 2.17

\[
\text{If } J \text{ is ordered by decreasing } \mathcal{P}\text{-priorities, i.e., } J_i > J_{i+1} \forall i, \text{ then it holds } \forall i \in [1, n] \text{ that} \\
\sum_{j=2}^{m} (\text{idle}^i_j - \text{idle}^i_{j-1}) \cdot s_{j-1} \geq \left( \text{idle}^i_m - \frac{\sum_{j=1}^{i-1} c_j}{s(1)} \right) \cdot s_1 \tag{2.28}
\]

Proof

The proof is based on the following fact: the term “\( \sum_{j=2}^{m} (\text{idle}^i_j - \text{idle}^i_{j-1}) \cdot s_{j-1} \)” denotes the cumulated amounts of execution units that can be executed in all the green areas (see Figure 2.17a on page 141) and this quantity is minimum if the following two conditions are satisfied, i.e., the worst-case is:

Cond 1. the distance between the idle-instants \( \text{idle}^i_1 \) and \( \text{idle}^i_m \) is minimum,

Cond 2. there is only a single green area located on the slowest processor \( \pi_1 \) and covering the whole distance between \( \text{idle}^i_1 \) and \( \text{idle}^i_m \).

This minimum quantity is illustrated on Figure 2.19, where these two conditions are satisfied. If we assume that the value of \( \text{idle}^i_m \) is known whereas that of \( \text{idle}^i_1 \) is unknown, then the first condition can be satisfied by considering an upper-bound on \( \text{idle}^i_1 \). From Lemma 2.15 on page 135, such an upper-bound on \( \text{idle}^i_1 \) is given by

\[
\frac{\text{idle}^i_1}{s(1)} \overset{\text{def}}{=} \frac{\sum_{j=1}^{i-1} c_j}{s(1)}
\]
and a lower-bound on the distance between \( \text{idle}_i^j \) and \( \text{idle}_m^i \) is thus given by

\[
\left( \text{idle}_m^i - \frac{\sum_{j=1}^{i-1} c_j}{s(1)} \right)
\]

Then, the second condition is satisfied by assuming that the green area is constituted of only 1 piece located on processor \( \pi_1 \). Under this assumption, the amount of execution units that can be executed within \( [\text{idle}_1^j, \text{idle}_m^i] \) is given by \( (\text{idle}_m^i - \frac{\sum_{j=1}^{i-1} c_j}{s(1)}) \cdot s_1 \) and the lemma follows.

Figure 2.19: The amount of execution units in the green area is a lower-bound on the amount of execution units that can be executed within \( [\text{idle}_1^j, \text{idle}_m^i] \) because (1) \( \overline{\text{idle}}_i^j \) is an upper-bound on \( \text{idle}_i^j \) and (2) \( s_1 \) is (one of) the slowest speed(s) of \( \pi \).
2.8 Validity test for uniform platforms and FJP schedulers

**Lemma 2.18**

If \( J \) is ordered by decreasing \( \mathcal{P} \)-priorities, i.e., \( J_i > J_{i+1} \ \forall i \), then an upper-bound on the completion time of job \( J_i \) is given by \( \text{comp}_i^{(2)}(J, \pi, \mathcal{P}) \), where (using our simplified notations)

\[
\text{comp}_i^{(2)} \overset{\text{def}}{=} \left( 1 - \frac{s_1}{s_m} \right) \cdot \text{idle}_m^i + \frac{1}{s_m} \left( c_i + s_1 \cdot \frac{\sum_{j=1}^{i-1} c_j}{s(1)} \right)
\] (2.29)

**Proof**

The proof directly follows from Lemmas 2.16 and 2.17. Indeed, we know from Lemma 2.16 that an upper-bound on the completion time of every job \( J_i \) is given by

\[
\text{comp}_i = \text{idle}_m^i - \sum_{j=2}^{m} \left( \text{idle}_m^j - \text{idle}_m^{j-1} \right) \cdot s_{j-1} + \frac{c_i}{s_m}
\]

and from Lemma 2.17 we have

\[
\sum_{j=2}^{m} \left( \text{idle}_m^j - \text{idle}_m^{j-1} \right) \cdot s_{j-1} \geq \left( \text{idle}_m^i - \frac{\sum_{j=1}^{i-1} c_j}{s(1)} \right) \cdot s_1
\]

Thus, it holds that

\[
\text{idle}_m^i - \sum_{j=2}^{m} \left( \text{idle}_m^j - \text{idle}_m^{j-1} \right) \cdot s_{j-1} + \frac{c_i}{s_m} \leq \text{idle}_m^i - \left( \text{idle}_m^i - \frac{\sum_{j=1}^{i-1} c_j}{s(1)} \right) \cdot s_1 + \frac{c_i}{s_m}
\]

\[
\leq \left( 1 - \frac{s_1}{s_m} \right) \cdot \text{idle}_m^i + \frac{1}{s_m} \left( c_i + s_1 \cdot \frac{\sum_{j=1}^{i-1} c_j}{s(1)} \right)
\]

and the instant

\[
\text{comp}_i^{(2)} \overset{\text{def}}{=} \left( 1 - \frac{s_1}{s_m} \right) \cdot \text{idle}_m^i + \frac{1}{s_m} \left( c_i + s_1 \cdot \frac{\sum_{j=1}^{i-1} c_j}{s(1)} \right)
\]

is therefore also an upper-bound on the completion time of \( J_i \).
Lemma 2.19

Let $\mathcal{P}$ denote any job priority assignment. Suppose that $J$ is ordered by decreasing $\mathcal{P}$-priorities, i.e., $J_i \succ_P J_{i+1}$ $\forall i$, and suppose that $J$ is scheduled by $\mathcal{P}$ upon $\pi$. If $\exists \ell$ such that

\[ J_\ell \text{ is not the lowest priority job according to } \mathcal{P}, \text{i.e., } \ell < n \]

and \[ \overline{\text{comp}}^{(2)}(J, \pi) = \max_{i=1}^{\ell} \overline{\text{comp}}^{(2)}(J, \pi, \mathcal{P}) \]

then there exists a job priority assignment $\mathcal{P}'$ such that

\[ J_\ell \text{ is the lowest priority job according to } \mathcal{P}' \]

and \[ \overline{\text{comp}}^{(2)}(J, \pi, \mathcal{P}') = \max_{i=1}^{\ell} \overline{\text{comp}}^{(2)}(J, \pi, \mathcal{P}') \] (2.30)

and \[ \overline{\text{comp}}^{(2)}(J, \pi, \mathcal{P}') \geq \overline{\text{comp}}^{(2)}(J, \pi) \] (2.31)

Proof

Let $\mathcal{P}'$ be the job priority assignment derived from $\mathcal{P}$ as follows:

\[ J_1 \succ_P J_2 \succ_P \cdots \succ_P J_{\ell-1} \succ_P J_{\ell+1} \succ_P \cdots \succ_P J_n \succ_P J_\ell \]

That is, $\mathcal{P}'$ is identical to $\mathcal{P}$ except that $\mathcal{P}'$ assigns the lowest priority to $J_\ell$. By hypothesis, we know that \[ \overline{\text{comp}}^{(2)}(J, \pi, \mathcal{P}) = \max_{i=1}^{\ell} \overline{\text{comp}}^{(2)}(J, \pi, \mathcal{P}) \]

where

\[ \overline{\text{comp}}^{(2)}(J, \pi, \mathcal{P}) \overset{\text{def}}{=} \left( 1 - \frac{s_1}{s_m} \right) \cdot \text{idle}_m(J_\ell, \pi, \mathcal{P}) + \frac{1}{s_m} \cdot \left( c_\ell + s_1 \cdot \sum_{j \succ_P J_\ell} c_j \right) \] (2.32)

according to Lemma 2.18. Therefore, assigning the lowest priority to $J_\ell$ as done by $\mathcal{P}'$ increases the number of jobs with a higher priority than $J_\ell$, hence increasing both the value of $\text{idle}_m(J_\ell, \pi, \mathcal{P})$ and the value of $\sum_{j \succ_P J_\ell} c_j$. Consequently, it holds that

\[ \overline{\text{comp}}^{(2)}(J, \pi, \mathcal{P}) \geq \overline{\text{comp}}^{(2)}(J, \pi, \mathcal{P}) \] (2.33)

thus providing Expression 2.31. Now, we can easily prove that Expression 2.30 also holds. Indeed, since $\mathcal{P}$ and $\mathcal{P}'$ assign the same priorities to the jobs $J_1, J_2, \ldots, J_{\ell-1}$, it follows from Expression 2.32 that $\forall r \in [1, \ldots, \ell-1], \ldots$
2.8 Validity test for uniform platforms and FJP schedulers

\begin{align}
\overline{\text{comp}}_r^{(2)}(J,\pi,P') &= \text{comp}_r^{(2)}(J,\pi,P) \\
\text{and } \forall r \in [\ell+1,\ldots,n],
\overline{\text{comp}}_r^{(2)}(J,\pi,P') &< \text{comp}_r^{(2)}(J,\pi,P)
\end{align}

(because job $J_{\ell}$ has the lowest priority according to $P'$ and thus, the expression of $\overline{\text{comp}}_r^{(2)}(J,\pi,P')$ considers one job less than the expression of $\text{comp}_r^{(2)}(J,\pi,P)$). In conclusion, it holds that

\begin{align}
\overline{\text{comp}}_r^{(2)}(J,\pi,P') &\leq \text{comp}_r^{(2)}(J,\pi,P) \forall J, \neq J_{\ell} \text{ (from Expressions 2.34 and 2.35)} \\
\text{and } \overline{\text{comp}}_r^{(2)}(J,\pi,P) &\leq \text{comp}_r^{(2)}(J,\pi,P) \forall J, \neq J_{\ell} \text{ (by hypothesis)} \\
\text{and } \text{comp}_r^{(2)}(J,\pi,P') &\leq \text{comp}_\ell^{(2)}(J,\pi,P') \text{ (from Expression 2.33)}
\end{align}

By transitivity, this yields

\begin{align}
\overline{\text{comp}}_r^{(2)}(J,\pi,P') &\leq \text{comp}_\ell^{(2)}(J,\pi,P') \forall J, \neq J_{\ell} \text{ and Expression 2.30 follows.}
\end{align}

**Corollary 2.6**

There always exists at least one job priority assignment $P$ such that $J_{\text{low}}$ is the lowest priority job according to $P$ and for every job priority assignment $\lambda$:

\[ \text{comp}_{\text{low}}^{(2)}(J,\pi,P) = \max_{i=1}^{n} \{ \text{comp}_{i}^{(2)}(J,\pi,\lambda) \} \]

i.e.,

\[ \text{comp}_{\text{low}}^{(2)}(J,\pi,P) \geq \text{maximum makespan} \]

**Proof**

The proof is a direct consequence of Lemma 2.19. Indeed, let $J_{\ell}$ be any job of $J$ and let $P'$ denote any job priority assignment such that

\[ \forall \lambda : \text{comp}_{\ell}^{(2)}(J,\pi,P') \geq \max_{i=1}^{n} \{ \text{comp}_{i}^{(2)}(J,\pi,\lambda) \} \]

i.e.,

\[ \text{comp}_{\ell}^{(2)}(J,\pi,P') \geq \text{maximum makespan} \]
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

If \( J_\ell \) is the lowest priority job according to \( P' \) then the corollary directly follows. Otherwise, suppose that \( J_\ell \) is not the lowest priority job according to \( P' \). If we define the priority assignment \( P \) as the same priority assignment as \( P' \) except that \( P \) assigns the lowest priority to \( J_\ell \), then we know from the proof of Lemma 2.19 that

\[
\overline{\text{comp}}^{(2)}(J, \pi, P) = \max_{i=1}^{n} \{ \text{comp}_i^{(2)}(J, \pi, P) \}
\]

and

\[
\overline{\text{comp}}^{(2)}(J, \pi, P) \geq \overline{\text{comp}}^{(2)}(J, \pi, P')
\]

and thus,

\[
\overline{\text{comp}}^{(2)}(J, \pi, P) \geq \text{maximum makespan}
\]

Lemma 2.20

There always exists at least one job priority assignment \( P^{\max} \) such that:

- \( J_{\max} \) is the lowest priority job according to \( P^{\max} \)
- \( c_{\max} = \max_{j=1}^{n} \{ c_j \} \)
- \( \overline{\text{comp}}^{(2)}(J, \pi, P^{\max}) \geq \text{maximum makespan} \)

Proof

The proof is made by \textit{contradiction}. Suppose that there does not exist such a job priority assignment \( P^{\max} \). Let \( P^{\text{other}} \) be any job priority assignment such that

- \( J_{\text{low}} \) is the lowest priority job,
- \( \overline{\text{comp}}^{(2)}_{\text{low}}(J, \pi, P^{\text{other}}) \geq \text{maximum makespan} \) (we know from Corollary 2.6 that such a priority assignment \( P^{\text{other}} \) exists), and
- \( J_{\text{low}} \) is not the (or one of the) job(s) with the largest processing time.

We show in the following that it is always possible to derive \( P^{\max} \) from \( P^{\text{other}} \), thus leading to a contradiction with our initial hypothesis. But first, let us introduce the following notations.
2.8 Validity test for uniform platforms and FJP schedulers

• Let $J_{\text{max}} \in J$ be the (or any) job of $J$ with the largest processing time.

• Let $P^{\text{max}}$ be the same job priority assignment as $P^{\text{other}}$, except that $P^{\text{max}}$ swaps the priority of $J_{\text{low}}$ and $J_{\text{max}}$, i.e., it assigns the lowest priority to the job $J_{\text{max}}$ of largest processing time.

• Recall that $\overline{\text{comp}}_{\text{low}}^{(2)}(J, \pi, P^{\text{other}})$ is an upper-bound on the completion time of the lowest priority job $J_{\text{low}}$ following $P^{\text{other}}$. According to Expression 2.29 (page 151), $\overline{\text{comp}}_{\text{low}}^{(2)}(J, \pi, P^{\text{other}})$ can be rewritten as

\[
\overline{\text{comp}}_{\text{low}}^{(2)}(J, \pi, P^{\text{other}}) \overset{\text{def}}{=} \mathcal{A}_{\text{low}}(J, \pi, P^{\text{other}}) + \mathcal{B}_{\text{low}}(J, \pi, P^{\text{other}}) \tag{2.36}
\]

where

\[
\mathcal{A}_{\text{low}}(J, \pi, P^{\text{other}}) \overset{\text{def}}{=} \left(1 - \frac{s_1}{s_m}\right) \cdot \text{idle}_m(J_{\text{low}}, \pi, P^{\text{other}}) \tag{2.37}
\]

and

\[
\mathcal{B}_{\text{low}}(J, \pi, P^{\text{other}}) \overset{\text{def}}{=} \frac{1}{s_m} \cdot \left( c_{\text{low}} + s_1 \cdot \frac{\sum_{J_{r \geq P^{\text{other}}}} c_r}{s(1)} \right) \tag{2.38}
\]

• Similarly, $\overline{\text{comp}}_{\text{max}}^{(2)}(J, \pi, P^{\text{max}})$ is an upper-bound on the completion time of the lowest priority job $J_{\text{max}}$ following $P^{\text{max}}$ and $\overline{\text{comp}}_{\text{max}}^{(2)}(J, \pi, P^{\text{max}})$ can be rewritten as

\[
\overline{\text{comp}}_{\text{max}}^{(2)}(J, \pi, P^{\text{max}}) \overset{\text{def}}{=} \mathcal{A}_{\text{max}}(J, \pi, P^{\text{max}}) + \mathcal{B}_{\text{max}}(J, \pi, P^{\text{max}}) \tag{2.39}
\]

and

\[
\mathcal{A}_{\text{max}}(J, \pi, P^{\text{max}}) \overset{\text{def}}{=} \left(1 - \frac{s_1}{s_m}\right) \cdot \text{idle}_m(J_{\text{max}}, \pi, P^{\text{max}}) \tag{2.40}
\]

and

\[
\mathcal{B}_{\text{max}}(J, \pi, P^{\text{max}}) \overset{\text{def}}{=} \frac{1}{s_m} \cdot \left( c_{\text{max}} + s_1 \cdot \frac{\sum_{J_{r \geq P^{\text{max}}}} c_r}{s(1)} \right) \tag{2.41}
\]

In the following, we measure the difference between $\overline{\text{comp}}_{\text{low}}^{(2)}(J, \pi, P^{\text{other}})$ and $\overline{\text{comp}}_{\text{max}}^{(2)}(J, \pi, P^{\text{max}})$ and we show that

\[
\overline{\text{comp}}_{\text{max}}^{(2)}(J, \pi, P^{\text{max}}) \geq \overline{\text{comp}}_{\text{low}}^{(2)}(J, \pi, P^{\text{other}})
\]
thus leading to a contradiction with our initial hypothesis following which such a job priority assignment $P_{\text{max}}$ does not exist. The remainder of the proof is divided into three parts. The first part measures the difference between $B_{\text{low}}(J, \pi, P_{\text{other}})$ and $B_{\text{max}}(J, \pi, P_{\text{max}})$, the second part measures the difference between $A_{\text{low}}(J, \pi, P_{\text{other}})$ and $A_{\text{max}}(J, \pi, P_{\text{max}})$ and finally, the third part asserts that $\text{comp}_{\text{low}}(J, \pi, P_{\text{max}}) \geq \text{comp}_{\text{low}}(J, \pi, P_{\text{other}})$.

Part 1. According to Expressions 2.38 and 2.41, the difference between $B_{\text{max}}(J, \pi, P_{\text{max}})$ and $B_{\text{low}}(J, \pi, P_{\text{other}})$ is given by:

\[
\begin{align*}
B_{\text{max}}(J, \pi, P_{\text{max}}) - B_{\text{low}}(J, \pi, P_{\text{other}}) &= \frac{1}{s_m} \left( c_{\text{max}} + s_1 \cdot \frac{\sum_{i > \text{max}} \max c_i}{s(1)} \right) - \frac{1}{s_m} \left( c_{\text{low}} + s_1 \cdot \frac{\sum_{i > \text{other}} \sum_{i > \text{low}} c_i}{s(1)} \right) \\
&= \frac{c_{\text{max}}}{s_m} + \frac{s_1 \cdot \sum_{i > \text{max}} \max c_i}{s(1) \cdot s_m} - \frac{c_{\text{low}}}{s_m} - \frac{s_1 \cdot \sum_{i > \text{other}} \sum_{i > \text{low}} c_i}{s(1) \cdot s_m} \\
&= \frac{c_{\text{max}} - c_{\text{low}}}{s_m} + \frac{s_1 \cdot (\sum_{i > \text{max}} \max c_i - \sum_{i > \text{other}} \sum_{i > \text{low}} c_i)}{s(1) \cdot s_m}
\end{align*}
\]

Since the jobs $J_{\text{max}}$ and $J_{\text{low}}$ have the lowest priority according to $P_{\text{max}}$ and $P_{\text{other}}$ (respectively) the above equality can be rewritten as

\[
B_{\text{max}}(J, \pi, P_{\text{max}}) - B_{\text{low}}(J, \pi, P_{\text{other}}) = \frac{c_{\text{max}} - c_{\text{low}}}{s_m} + \frac{s_1 \cdot (c_{\text{low}} - c_{\text{max}})}{s(1) \cdot s_m} = \frac{(s(1) - s_1) \cdot (c_{\text{max}} - c_{\text{low}})}{s_m \cdot s(1)}
\]

Part 2. This part measures the difference between the terms $A_{\text{low}}(J, \pi, P_{\text{other}})$ and $A_{\text{max}}(J, \pi, P_{\text{max}})$. But first, recall that in the function $A_{\text{low}}(J, \pi, P_{\text{other}})$ defined in Expression 2.37, the term $\text{idlem}(J_{\text{low}}, \pi, P_{\text{other}})$ denotes the $m^{\text{th}}$ idle-instant in the schedule of every job with a higher priority than $J_{\text{low}}$ following $P_{\text{other}}$, i.e., the schedule of $J \setminus \{J_{\text{low}}\}$. Similarly, in the function $A_{\text{max}}(J, \pi, P_{\text{max}})$ defined in Expression 2.40, the term $\text{idlem}(J_{\text{max}}, \pi, P_{\text{max}})$ denotes the $m^{\text{th}}$ idle-instant in the schedule of every job with a higher priority than $J_{\text{max}}$ following $P_{\text{max}}$, i.e., the schedule of $J \setminus \{J_{\text{max}}\}$. Since $c_{\text{max}} > c_{\text{low}}$, we can consider that the set of jobs $J \setminus \{J_{\text{low}}\}$ is the same set of jobs as $J \setminus \{J_{\text{max}}\}$ except that the processing time $c_{\text{low}}$ of job $J_{\text{low}}$ (present in $J \setminus \{J_{\text{max}}\}$) has been increased to $c_{\text{max}}$ in $J \setminus \{J_{\text{low}}\}$. Therefore, it holds from Lemma 2.6 (page 109) that
2.8 Validity test for uniform platforms and FJP schedulers

\[
\text{idle}_m(j_{\text{low}}, \pi, \mathcal{P}^{\text{other}}) \leq \text{idle}_m(j_{\text{max}}, \pi, \mathcal{P}^{\text{max}}) + \frac{c_{\text{max}} - c_{\text{low}}}{s_m} \tag{2.42}
\]

As a result, according to Expression 2.37 and 2.40, the difference between \(\mathcal{A}_{\text{max}}(j, \pi, \mathcal{P}^{\text{max}})\) and \(\mathcal{A}_{\text{low}}(j, \pi, \mathcal{P}^{\text{other}})\) is given by:

\[
\mathcal{A}_{\text{max}}(j, \pi, \mathcal{P}^{\text{max}}) - \mathcal{A}_{\text{low}}(j, \pi, \mathcal{P}^{\text{other}}) = \left(1 - \frac{s_1}{s_m}\right) \cdot \text{idle}_m(j_{\text{max}}, \pi, \mathcal{P}^{\text{max}}) - \left(1 - \frac{s_1}{s_m}\right) \cdot \text{idle}_m(j_{\text{low}}, \pi, \mathcal{P}^{\text{other}})
\]

and from Inequality 2.42, the above equality can be rewritten as

\[
\mathcal{A}_{\text{max}}(j, \pi, \mathcal{P}^{\text{max}}) - \mathcal{A}_{\text{low}}(j, \pi, \mathcal{P}^{\text{other}}) \geq \left(1 - \frac{s_1}{s_m}\right) \cdot \left(\text{idle}_m(j_{\text{max}}, \pi, \mathcal{P}^{\text{max}}) - \text{idle}_m(j_{\text{low}}, \pi, \mathcal{P}^{\text{other}})\right) - \frac{c_{\text{max}} - c_{\text{low}}}{s_m}
\]

Multiplying both sides by \((-1)\) yields

\[
\mathcal{A}_{\text{low}}(j, \pi, \mathcal{P}^{\text{other}}) - \mathcal{A}_{\text{max}}(j, \pi, \mathcal{P}^{\text{max}}) \leq \frac{\left(1 - \frac{s_1}{s_m}\right) \cdot \frac{c_{\text{max}} - c_{\text{low}}}{s_m}}{s_m} \leq \frac{(s_m - s_1) \cdot (c_{\text{max}} - c_{\text{low}})}{s_m^2}
\]

**Part 3.** This third part asserts that \(\overline{\text{comp}}_{\text{max}}^{(2)}(j, \pi, \mathcal{P}^{\text{max}}) \geq \overline{\text{comp}}_{\text{low}}^{(2)}(j, \pi, \mathcal{P}^{\text{other}})\). Recall that from Parts 1 and 2 we have:

\[
\mathcal{B}_{\text{max}}(j, \pi, \mathcal{P}^{\text{max}}) = \mathcal{B}_{\text{low}}(j, \pi, \mathcal{P}^{\text{other}}) + \frac{(s(1) - s_1) \cdot (c_{\text{max}} - c_{\text{low}})}{s_m \cdot s(1)} \tag{2.43} \quad \text{from Part 1}
\]

\[
\mathcal{A}_{\text{max}}(j, \pi, \mathcal{P}^{\text{max}}) \geq \mathcal{A}_{\text{low}}(j, \pi, \mathcal{P}^{\text{other}}) - \frac{(s_m - s_1) \cdot (c_{\text{max}} - c_{\text{low}})}{s_m^2} \tag{2.44} \quad \text{from Part 2}
\]

From the definitions of \(\overline{\text{comp}}_{\text{max}}^{(2)}(j, \pi, \mathcal{P}^{\text{max}})\) and \(\overline{\text{comp}}_{\text{low}}^{(2)}(j, \pi, \mathcal{P}^{\text{other}})\) in Expression 2.39 and 2.36 (respectively), we have

\[
\overline{\text{comp}}_{\text{max}}^{(2)}(j, \pi, \mathcal{P}^{\text{max}}) - \overline{\text{comp}}_{\text{low}}^{(2)}(j, \pi, \mathcal{P}^{\text{other}}) = \mathcal{A}_{\text{max}}(j, \pi, \mathcal{P}^{\text{max}}) + \mathcal{B}_{\text{max}}(j, \pi, \mathcal{P}^{\text{max}}) - \mathcal{A}_{\text{low}}(j, \pi, \mathcal{P}^{\text{other}}) - \mathcal{B}_{\text{low}}(j, \pi, \mathcal{P}^{\text{other}})
\]
By replacing $B_{max}(J, \pi, p^{max})$ with $B_{low}(J, \pi, p^{other}) + \frac{(s(1) - s_1) \cdot (c_{max} - c_{low})}{s_m \cdot s(1)}$ according to Equality 2.43, we get

$$\text{comp}^{(2)}_{max}(J, \pi, p^{max}) - \text{comp}^{(2)}_{low}(J, \pi, p^{other}) = A_{max}(J, \pi, p^{max}) - A_{low}(J, \pi, p^{other}) + \frac{(s(1) - s_1) \cdot (c_{max} - c_{low})}{s_m \cdot s(1)}$$

and replacing $A_{max}(J, \pi, p^{max})$ with $A_{low}(J, \pi, p^{other}) - \frac{(s_{m} - s_1) \cdot (c_{max} - c_{low})}{s_m}$ according to Inequality 2.44 yields

$$\text{comp}^{(2)}_{max}(J, \pi, p^{max}) - \text{comp}^{(2)}_{low}(J, \pi, p^{other}) \geq \frac{(s_{m} - s_1) \cdot (c_{max} - c_{low})}{s_m \cdot s(1)}$$

Since $s(1) \geq s_m$ and $c_{max} \geq c_{low}$ (because $c_{max}$ is the maximal processing time), the above inequality can be rewritten as

$$\text{comp}^{(2)}_{max}(J, \pi, p^{max}) - \text{comp}^{(2)}_{low}(J, \pi, p^{other}) \geq 0$$

In conclusion, $p^{max}$ is a job priority assignment that assigns the lowest priority to the job $J_{max}$ with the largest processing time and we showed in this third part that

$$\text{comp}^{(2)}_{max}(J, \pi, p^{max}) \geq \text{comp}^{(2)}_{low}(J, \pi, p^{other})$$

Since by hypothesis we know that $\text{comp}^{(2)}_{low}(J, \pi, p^{other})$ is an upper-bound on the maximum makespan, it holds that $\text{comp}^{(2)}_{max}(J, \pi, p^{max})$ is also an upper-bound on the maximum makespan. This contradicts our initial hypothesis following which such a job priority assignment $p^{max}$ does not exists.
2.8 Validity test for uniform platforms and FJP schedulers

Lemma 2.21

Suppose that \( c_1 \leq c_2 \leq \cdots \leq c_n \). If \( J \) is scheduled upon \( \pi \) by any global, FJP and strongly work-conserving scheduler, then an upper-bound \( ms_{\text{unif}}^2(J, \pi) \) on the makespan is given by \( \hat{t}_n \), where \( \hat{t}_n \) is computed by the following iterative process.

\[
\begin{aligned}
\hat{t}_1 &= \frac{c_1}{s_m} \quad \text{(initialization)} \\
\hat{t}_{i+1} &= \left( 1 - \frac{c_i}{s_m} \right) \cdot \hat{t}_i + \frac{1}{s_m} \cdot \left( c_{i+1} + s_1 \cdot \frac{\sum_{j=1}^{i} c_j}{s(1)} \right) \quad \text{(iterative step)}
\end{aligned}
\]

Proof

From Lemma 2.20, we know that there exists at least one job priority assignment \( \mathcal{P}^{max} \) that assigns the lowest priority to the job \( J_n \) with the largest processing time and such that

\[
\text{comp}^{(2)}(J, \pi, \mathcal{P}^{max}) \geq \text{maximum makespan}
\]

From Expression 2.29 (page 151) we know that

\[
\text{comp}^{(2)}(J, \pi, \mathcal{P}^{max}) \overset{\text{def}}{=} \left( 1 - \frac{s_1}{s_m} \right) \cdot \text{idle}_m(J_n, \pi, \mathcal{P}^{max}) + \frac{1}{s_m} \cdot \left( c_n + s_1 \cdot \frac{\sum_{j=1}^{i} c_j}{s(1)} \right) \tag{2.45}
\]

and it obviously holds \( \forall i \in [1, n] \) that the exact makespan generated by the set of jobs \( J_j \succ \mathcal{P}^{max} J_i \) cannot be larger than the maximum completion time of these jobs \( J_j \succ \mathcal{P}^{max} J_i \). That is, it holds \( \forall i \in [1, n] \) that

\[
\text{idle}_m(J_i, \pi, \mathcal{P}^{max}) \leq \max_{J_j \succ \mathcal{P}^{max} J_i} \left\{ \text{comp}^{(2)}(J, \pi, \mathcal{P}^{max}) \right\}
\]

Consequently, Expression 2.45 can be rewritten as

\[
\text{comp}^{(2)}(J, \pi, \mathcal{P}^{max}) \leq \left( 1 - \frac{s_1}{s_m} \right) \cdot \max_{J_j \succ \mathcal{P}^{max} J_i} \left\{ \text{comp}^{(2)}(J, \pi, \mathcal{P}^{max}) \right\} + \frac{1}{s_m} \cdot \left( c_n + s_1 \cdot \frac{\sum_{j=1}^{i} c_j}{s(1)} \right) \tag{2.46}
\]

and an upper-bound on \( \text{comp}^{(2)}(J, \pi, \mathcal{P}^{max}) \) is thus given by \( \hat{t}_n \), where

\[
\hat{t}_n \overset{\text{def}}{=} \left( 1 - \frac{s_1}{s_m} \right) \cdot \max_{J_j \succ \mathcal{P}^{max} J_i} \left\{ \text{comp}^{(2)}(J, \pi, \mathcal{P}^{max}) \right\} + \frac{1}{s_m} \cdot \left( c_n + s_1 \cdot \frac{\sum_{j=1}^{i} c_j}{s(1)} \right) \tag{2.47}
\]
Considering the subset \( J' = \{ J_1, J_2, \ldots, J_{n-1} \} \) of jobs, we know from Lemma 2.20 that there exists at least one job priority assignment \( P' \) that assigns the lowest priority to the job \( J_{n-1} \) with the largest processing time and such that \( \forall X \)

\[
\text{comp}^{(2)}_{n-1}(J', \pi, P') = \max_{i=1}^{n-1} \{\text{comp}^{(2)}_i(J', \pi, X)\}
\]

i.e., \( \text{comp}^{(2)}_{n-1}(J', \pi, P') \) is an upper-bound on the makespan for the set \( J' \) of jobs, while considering every job priority assignments \( X \). Thereby, if \( P_{\text{max}} \) assigns a priority to \( J_{n-1} \) such that \( \forall i \in [1, n-2], J_i > P_{\text{max}} J_{n-1} > P_{\text{max}} J_i \)

then we get

\[
\text{comp}^{(2)}_{n-1}(J', \pi, P_{\text{max}}) = \max_{i=1}^{n-1} \{\text{comp}^{(2)}_i(J', \pi, P_{\text{max}})\}
\]

and Expression 2.47 can be rewritten as

\[
\hat{t}_n \overset{\text{def}}{=} \left(1 - \frac{s_1}{s_m}\right) \cdot \text{comp}^{(2)}_{n-1}(J', \pi, P_{\text{max}}) + \frac{1}{s_m} \cdot \left(c_n + s_1 \cdot \frac{\sum_{j=1}^{n} c_j}{s(1)}\right)
\]

The same reasoning as the one above can be used iteratively in order to determine an upper-bound on \( \text{comp}^{(2)}_{n-1}(J', \pi, P_{\text{max}}) \). Ultimately, this iterative development causes \( P_{\text{max}} \) to imitate the job priority assignment SJF (Shortest Job First) and an upper-bound \( \overline{m_{2,\text{unif}}}(J, \pi) \) on the makespan can therefore be computed by the iterative process:

\[
\begin{aligned}
\hat{t}_1 &= \frac{c_1}{s_m} \quad \text{(initialization)} \\
\hat{t}_{i+1} &= \left(1 - \frac{s_1}{s_m}\right) \cdot \hat{t}_i + \frac{1}{s_m} \cdot \left(c_{i+1} + s_1 \cdot \frac{\sum_{j=1}^{i} c_j}{s(1)}\right) \quad \text{(iterative step)}
\end{aligned}
\]

That is, even though we know that scheduling the jobs according to SJF does not lead to the maximum makespan (as shown in the counterexample of Section 2.8.1, page 129), this policy leads nevertheless to an upper-bound \( \overline{m_{2,\text{unif}}}(J, \pi) \) on the upper-bounds \( \text{comp}^{(2)}_{n}(J, \pi, X) \) \( \forall X \). The lemma follows.

**Lemma 2.22**

Suppose that \( c_1 \leq c_2 \leq \cdots \leq c_n \). If \( J \) is scheduled upon \( \pi \) by any FJP job priority assignment then an upper-bound \( \overline{m_{2,\text{unif}}}(J, \pi) \) on the makespan is given by
2.8 Validity test for uniform platforms and FJP schedulers

\[
\overline{m_{\text{unif}}}^{\text{unif}}(J, \pi) \overset{\text{def}}{=} \frac{1}{s_m} \cdot \sum_{i=1}^{n} \left( c_i + s_1 \cdot \frac{\sum_{j=1}^{i-1} c_j}{s(1)} \right) \cdot K_{n-i}
\]  

(2.48)

where \( K_j \) is such that \( \forall j, \)

\[
K_j \overset{\text{def}}{=} \begin{cases} 
1 & \text{if } s_1 = s_m \text{ and } j = 0 \\
(1 - \frac{s_1}{s_m})^j & \text{otherwise}
\end{cases}
\]

Proof

This proof consists in rewriting the recursive process given in Lemma 2.21 as a non-recursive function. First, we rewrite the iterative process as follows.

\[
\begin{align*}
\hat{t}_1 &= \frac{c_1}{s_m} \quad \text{(initialization)} \\
\hat{t}_i &= K \cdot \hat{t}_{i-1} + G_i \quad \text{(iterative step)}
\end{align*}
\]

where \( K \overset{\text{def}}{=} (1 - \frac{s_1}{s_m}) \) and \( G_i \overset{\text{def}}{=} \frac{1}{s_m} \left( c_i + s_1 \cdot \frac{\sum_{j=1}^{i-1} c_j}{s(1)} \right) \). From the iterative step of this simplified process, we have \( \forall i, 2 \leq i \leq n, \)

\[
\hat{t}_i = K \cdot \hat{t}_{i-1} + G_i = K \cdot (K \cdot \hat{t}_{i-2} + G_{i-1}) + G_i = K \cdot (K \cdot (K \cdot \hat{t}_{i-3} + G_{i-2}) + G_{i-1}) + G_i = K \cdot (K \cdot (K \cdot \cdots K \cdot (\hat{t}_1 + G_2) + \cdots) + G_{i-2}) + G_{i-1} + G_i \quad (2.49)
\]

\[
K^{i-1} \cdot \hat{t}_1 + \sum_{j=0}^{i-2} K^j \cdot G_{i-j} \quad (2.50)
\]

Notice that in the sum above, we get the term \( K^0 + G_i \) when \( j = 0 \) and according to the definition of \( K \overset{\text{def}}{=} (1 - \frac{s_1}{s_m}) \), in the particular case where \( s_1 = s_m \) (i.e., the platform is identical), we get \( K^0 = (1 - 1)^0 \) which is undetermined. Thereby, passing from Expression 2.49 to Expression 2.50 requires to redefine \( K \) beforehand, i.e., we define \( K_j \forall j \) such that

\[
K_j \overset{\text{def}}{=} \begin{cases} 
1 & \text{if } s_1 = s_m \text{ and } j = 0 \\
(1 - \frac{s_1}{s_m})^j & \text{otherwise}
\end{cases}
\]

(2.51)
and Equality 2.50 can be rewritten as
\[
\hat{t}_i = K_{i-1} \cdot \hat{t}_1 + \sum_{j=0}^{i-2} K_j \cdot G_{i-j}
\]

Let \( \ell \) def \( = i - j \), the previous equality yields
\[
\hat{t}_i = K_{i-1} \cdot \hat{t}_1 + \sum_{\ell=2}^{i} K_{i-\ell} \cdot G_{\ell}
\]

\[
= K_{i-1} \cdot \frac{c_1}{s_m} + \sum_{\ell=2}^{i} K_{i-\ell} \cdot \frac{1}{s_m} \cdot \left( c_{\ell} + s_1 \cdot \frac{\sum_{j=1}^{\ell-1} c_j}{s(1)} \right)
\]

\[
= \frac{1}{s_m} \cdot \left( c_1 \cdot K_{i-1} + \sum_{\ell=2}^{i} \left( c_{\ell} + s_1 \cdot \frac{\sum_{j=1}^{\ell-1} c_j}{s(1)} \right) \cdot K_{i-\ell} \right)
\]

\[
= \frac{1}{s_m} \cdot \sum_{\ell=1}^{i} \left( c_{\ell} + s_1 \cdot \frac{\sum_{j=1}^{\ell-1} c_j}{s(1)} \right) \cdot K_{i-\ell} \quad \text{since} \quad s_1 \cdot \frac{\sum_{j=1}^{\ell-1} c_j}{s(1)} = 0 \quad \text{for} \quad \ell = 1
\]

That is, for \( i = n \) we have
\[
\hat{t}_n = \frac{1}{s_m} \cdot \sum_{\ell=1}^{n} \left( c_{\ell} + s_1 \cdot \frac{\sum_{j=1}^{\ell-1} c_j}{s(1)} \right) \cdot K_{n-\ell}
\]

(2.52)

where \( K \) is such as defined in Expression 2.51. The lemma follows from the fact that \( \overline{m_2^{\text{unif}}}(J, \pi) \) def \( = \hat{t}_n \) from Lemma 2.21.

From Lemma 2.22, a sufficient validity test can therefore be formalized as follows for the protocol SM-MSO.

**Validity Test 2.4 (For SM-MSO, FJP schedulers and uniform platforms)**

For any multimode real-time application \( \tau \) and any identical platform \( \pi \) composed of \( m \) processors, the protocol SM-MSO is valid provided that, for every mode \( M^i \),

\[
\overline{m_2^{\text{unif}}}(J^w, \pi) \leq \min_{j \neq i} \left\{ \frac{1}{n_i} \min_{k=1}^{n_i} \{ D_k^j(M^i) \} \right\}
\]
where $J^{w_{c}}_{i}$ is the worst-case rem-job set issued from mode $M_{i}$. This set contains $n_{i}$ jobs of processing time $C^{i}_{1}, C^{i}_{2}, \ldots, C^{i}_{n_{i}}$ and is ordered by non-decreasing job processing times, i.e., $C^{i}_{j} \geq C^{i}_{j-1}$ $\forall j = 2, 3, \ldots, n_{i}$.

### 2.8.6 A third upper-bound on the makespan

Here, we follow exactly the same seven steps as those described in the previous section, but we determine another lower-bound on the expression

$$\sum_{j=2}^{m} (idle_{j}(J_{i}, \pi, P) - idle_{j-1}(J_{i}, \pi, P)) \cdot s_{j-1}$$

in the first step. Consequently, we have to “re-prove” the seven steps considering this alternative lower-bound. However, due to the similarity between the proofs of the previous section and those related to this third upper-bound $\bar{m_{s}}_{3}^{\text{unif}}(\tau, \pi)$, we only present here the formal statements of the lemmas and corollaries. All the proofs are given in the Appendix (page 351). This development starts with the following definition.

**Definition 2.16 (Critical speed $s_{x}$)**

For any uniform platform $\pi = [s_{1}, s_{2}, \ldots, s_{m}]$ composed of $m$ processors, we say that any speed $s_{x}$, $1 \leq x \leq m$, is critical if and only if

$$x = \arg\min_{i \in [1,m]} \left( \frac{s_{i}}{\sum_{j=1}^{i} s_{j}} \right)$$

Then, the following lemmas and corollaries follows the seven steps introduced previously. As in the previous section and for the same reasons, we use in the following statements the notations $\text{idle}_{j}$ and $\text{comp}_{j}$ and $\text{comp}_{3}$ instead of $idle_{j}(J_{i}, \pi, P)$, $\text{comp}_{j}(J_{i}, \pi, P)$ and $\text{comp}_{3}(J_{i}, \pi, P)$, respectively.

**Lemma 2.23**

If $J$ is ordered by decreasing $P$-priorities, i.e., $J_{i} > J_{i+1}$ $\forall i$, then it holds $\forall i \in [1, n]$ that
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

\[
\sum_{j=2}^{m} (\text{id}le_j^i - \text{id}le_{j-1}^i) \cdot s_{j-1} \geq \left( \text{id}le_m^i - \frac{\sum_{j=1}^{i-1} c_j^i}{s(1)} \right) \cdot s_m \cdot \frac{s_x}{\sum_{j=1}^{x} s_j}
\]

where \( s_x \) denotes any critical speed of \( \pi \).

Lemma 2.24

If \( J \) is ordered by decreasing \( P \)-priorities, i.e., \( J_i > J_{i+1} \) \( \forall i \), then an upper-bound on the completion time of job \( J_i \) is given by \( \overline{\text{comp}}^3_{i}(J, \pi, P) \), where (using our simplified notations)

\[
\overline{\text{comp}}^3_{i}(J, \pi, P) \overset{\text{def}}{=} \left( 1 - \frac{s_x}{\sum_{j=1}^{x} s_j} \right) \cdot \text{id}le_m^i + \left( \frac{c_i}{s_m} + \frac{\sum_{j=1}^{i-1} c_j^i}{s(1) \cdot \sum_{j=1}^{x} s_j} \right)
\] \quad (2.53)

Lemma 2.25

Let \( P \) denote any job priority assignment. Suppose that \( J \) is ordered by decreasing \( P \)-priority, i.e., \( J_i \succ_P J_{i+1} \) \( \forall i \), and suppose that \( J \) is scheduled by \( P \) upon \( \pi \). If \( \exists \ell \in [1, n] \) such that

\[
\overline{\text{comp}}^3_{\ell}(J, \pi, P) = \max_{i=1}^{n} \{ \overline{\text{comp}}^3_{i}(J, \pi, P) \}
\]

then we can always derive another job priority assignment \( P' \) from \( P \) such that

- \( J_\ell \) is the lowest priority job according to \( P' \)
- \( \overline{\text{comp}}^3_{\ell}(J, \pi, P') = \max_{i=1}^{n} \{ \overline{\text{comp}}^3_{i}(J, \pi, P') \} \) \quad (2.54)
- \( \overline{\text{comp}}^3_{\ell}(J, \pi, P') \geq \overline{\text{comp}}^3_{\ell}(J, \pi, P) \) \quad (2.55)

Corollary 2.7

There always exists at least one job priority assignment \( P \) such that \( J_{\text{low}} \) is the lowest priority job according to \( P \) and for every job priority assignment \( X \):

164
2.8 Validity test for uniform platforms and FJP schedulers

\[
\text{comp}_{\text{low}}^{(3)} (J, \pi, \mathcal{P}) = \max_{i=1}^{n} \{ \text{comp}_{i}^{(3)} (J, \pi, \mathcal{X}) \}
\]

i.e.,

\[
\text{comp}_{\text{low}}^{(3)} (J, \pi, \mathcal{P}) \geq \text{maximum makespan}
\]

Lemma 2.26

There always exists at least one job priority assignment \( \mathcal{P}^{\text{max}} \) such that:

- \( J_{\text{max}} \) is the lowest priority job according to \( \mathcal{P}^{\text{max}} \)
- \( c_{\text{max}} = \max_{j=1}^{n} \{ c_{j} \} \)
- \( \text{comp}_{\text{low}}^{(3)} (J, \pi, \mathcal{P}^{\text{max}}) \geq \text{maximum makespan} \)

Lemma 2.27

Suppose that \( c_{1} \leq c_{2} \leq \cdots \leq c_{n} \). If \( J \) is scheduled upon \( \pi \) by any global, FJP and strongly work-conserving scheduler, then an upper-bound \( \text{ms}_{\text{unif}}^{3} (J, \pi) \) on the makespan is given by \( \hat{t}_{n} \), where \( \hat{t}_{n} \) is computed by the following iterative process.

\[
\begin{align*}
\hat{t}_{1} &= \frac{c_{1}}{s_{m}} \quad \text{(initialization)} \\
\hat{t}_{i+1} &= \left( 1 - \frac{s_{x}}{\sum_{j=1}^{x} s_{j}} \right) \cdot \hat{t}_{i} + \left( \frac{c_{i}}{s_{m}} + s_{x} \cdot \frac{\sum_{j=1}^{i-1} c_{j}}{s(1) \cdot \sum_{j=1}^{x} s_{j}} \right) \quad \text{(iterative step)}
\end{align*}
\]

Lemma 2.28

Suppose that \( c_{1} \leq c_{2} \leq \cdots \leq c_{n} \). If \( J \) is scheduled upon \( \pi \) by any FJP job priority assignment then an upper-bound \( \text{ms}_{\text{unif}}^{3} (J, \pi) \) on the makespan is given by
\[ ms_{3}^{\text{unif}}(J, \pi) \text{ def } = \frac{1}{s} \sum_{\ell=1}^{n} \left( c_{\ell} + \frac{s_{x} \cdot s_{m}}{s(1) \cdot \sum_{j=1}^{\ell-1} s_{j}} \cdot \sum_{j=1}^{\ell-1} c_{j} \right) \cdot H_{n-\ell} \] 

where \( H_{j} \) is such that \( \forall j, \)

\[ H_{j} \text{ def } = \begin{cases} 
1 & \text{if } s_{x} = \sum_{i=1}^{x} s_{i} \text{ and } j = 0 \\
\left(1 - \frac{s_{x}}{\sum_{i=1}^{x} s_{i}}\right)^{j} & \text{otherwise}
\end{cases} \]

From Lemma 2.28, a "sufficient" validity test can therefore be formalized as follows for the protocol SM-MSO.

**Validity Test 2.5 (For SM-MSO, FJP schedulers and uniform platforms)**

For any multimode real-time application \( \tau \) and any identical platform \( \pi \) composed of \( m \) processors, the protocol SM-MSO is valid provided that, for every mode \( M_{i} \),

\[ ms_{3}^{\text{unif}}(J^{\text{wc}}_{i}, \pi) \leq \min_{j \neq i} \left\{ \min_{k=1}^{n_{i}} D_{k}^{j}(M_{i}) \right\} \]

where \( J^{\text{wc}}_{i} \) is the worst-case rem-job set issued from mode \( M_{i} \). This set contains \( n_{i} \) jobs of processing time \( C_{1}^{i}, C_{2}^{i}, \ldots, C_{n_{i}}^{i} \) and is ordered by non-decreasing job processing times, i.e., \( C_{j}^{i} \geq C_{j-1}^{i} \forall j = 2, 3, \ldots, n_{i} \).

The accuracy of each upper-bound \( ms_{\text{ident}}^{3}(J, \pi), ms_{1}^{\text{unif}}(J, \pi), ms_{2}^{\text{unif}}(J, \pi) \) and \( ms_{3}^{\text{unif}}(J, \pi) \) is studied in Section 2.11 on page 177. Then, Section 2.12 (page 187) compares the efficiency of these upper-bounds through simulations.

### 2.9 Validity test for uniform platforms and FTP schedulers

#### 2.9.1 Determination of upper-bounds on the idle-instants

This section follows the same reasoning as the one followed for identical platforms and FTP schedulers. For any transition from a specific mode \( M_{i} \) to any other mode \( M_{j} \),
2.9 Validity test for uniform platforms and FTP schedulers

the knowledge of the worst-case rem-job set \( J_i^{wc} \) and the fact that the priorities are known beforehand enable to compute the exact maximum idle-instants \( \text{idle}_k, 1 \leq k \leq m \), simply by simulating the scheduling of the worst-case rem-job set and by measuring the idle-instants \( \text{idle}_k, 1 \leq k \leq m \), in that schedule (from Corollary 2.1 presented on page 107). Thus, each idle-instant \( \text{idle}_k \) measured in the schedule of the worst-case rem-job set is an upper-bound on the idle-instants \( \text{idle}_k \) in the schedule derived from any other set of rem-jobs. In conclusion, FTP schedulers enable to determine the exact maximum idle-instants \( \text{idle}_k, 1 \leq k \leq m \), rather than over-approximating them (as done for the FJP schedulers). In Lemma 2.29 below, we provide the exact values of \( \text{idle}_j(J_i, \pi, \mathcal{P}) \) \( \forall j \in [1, m], i \in [1, n] \) and \( \forall \mathcal{P} \), assuming that every job \( J_i \) executes for its WCET. However, in this particular case of FTP scheduler, we redefine the idle-instants \( \text{idle}_j(J_i, \pi, \mathcal{P}) \) as follows

**Definition 2.17 (Idle-instant \( \text{idle}_j(J_i, \pi, \mathcal{P}) \))**

If \( S^i \) denotes the schedule upon \( \pi \) of only the jobs with a higher (or equal) priority than \( J_i \) according to \( \mathcal{P} \), then \( \text{idle}_j(J_i, \pi, \mathcal{P}) \) is the earliest instant in \( S^i \) at which at least \( j \) processors idle.

The only difference between this definition and the previous one (given on page 140) resides in the “higher (or equal) priority than (...)”. The reason for this redefinition is that, with the previous one, it was not possible to express the idle-instants \( \text{idle}_j(J_i, \pi, \mathcal{P}) \) (for \( j = 1, \ldots, m \)) as defined in Definition 2.11 (page 94). Indeed, these idle-instants \( \text{idle}_j(J_i, \pi, \mathcal{P}) \) consider that every job of \( J \) are scheduled while the previous definition of the idle-instants \( \text{idle}_j(J_i, \pi, \mathcal{P}) \) requires a job index \( i \) and considers that only the jobs with a higher priority than \( J_i \) are scheduled. Thereby, this previous definition always excludes the job \( J_i \) in the computation of the idle-instants. Now, thanks to this new definition, the idle-instants \( \text{idle}_j(J_i, \pi, \mathcal{P}) \) (for \( j = 1, \ldots, m \)) can be expressed by \( \text{idle}_j(J_{low}, \pi, \mathcal{P}) \), where \( J_{low} \) is the lowest priority job according to \( \mathcal{P} \). Once again, we use in the following Corollary 2.8 the notations \( \text{idle}_j \) to refer to the idle-instants \( \text{idle}_j(J_i, \pi, \mathcal{P}) \) defined as in Definition 2.17.

---

1 Exact in the sense that this value is actually reached if every job executes for its WCET.
Lemma 2.29 (From P. Meumeu Yomsi, V. Nelis and J. Goossens [30])

Let $\pi = [s_1, s_2, \ldots, s_m]$ denote any uniform multiprocessor platform composed of $m$ processors and assume that $s_i \geq s_{i-1}$, $\forall i = 2, 3, \ldots, m$. Let $J = \{J_1, J_2, \ldots, J_n\}$ be any set of $n$ jobs, all released at time $t = 0$, with respective computation time $c_1, c_2, \ldots, c_n$. Let $S$ denote any global, FTP and strongly work-conserving scheduler and suppose that $J$ is ordered by decreasing $S$-priority, i.e., $J_i \succ_S J_{i+1}$. If these jobs are scheduled by $S$ upon $\pi$, then idle$^0_j$ is inductively defined as follows: (initialization) idle$^0_j \overset{\text{def}}{=} 0$, $\forall j \leq m$ and idle$^{j+1}_m \overset{\text{def}}{=} \infty$, $\forall i$, (iteration)

\[
\text{idlle}^j_i = \begin{cases} 
\text{idlle}^{j-1}_{i+1} & \text{if } \text{idlle}^{j-1}_i = \text{idlle}^{j-1}_{i+1} \\
\text{idlle}^{j-1}_{i+1} & \text{else if } c_i \geq \sum_{k=1}^{j} (\text{idlle}^{j-1}_k - \text{idlle}^{j-1}_k) \cdot s_k \\
\text{idlle}^{j-1}_i + \left( c_i - \sum_{k=1}^{j-1} (\text{idlle}^{j-1}_k - \text{idlle}^{j-1}_k) \cdot s_k \right) / s_i & \text{otherwise}
\end{cases}
\]  

(2.57)

Proof

Initially, the $m$ processors idle and thus, idle$^0_j = 0$, $\forall j$, $1 \leq j \leq m$. We find convenient to define idle$^{j+1}_m \overset{\text{def}}{=} \infty$, $\forall i$, which means that we have at most $m$ processors available. In the following, we prove the correctness of the value of idle$^j_i$ ($\forall j, 1 \leq j \leq m$) assuming that idle$^{j-1}_i$ are defined ($\forall j \leq m + 1$). The idle-instants idle$^{j-1}_i$ define a staircase as illustrated in Figure 2.20 for the scheduling of jobs $J_1, \ldots, J_{m-1}$. Thus, job $J_i$ can only progress into the blue areas and two cases have to be distinguished:

**Case 1.** idle$^{j-1}_j = \text{idlle}^{j-1}_{j+1}$, meaning that at least one processor faster than $\pi_j$ becomes available at time idle$^{j-1}_j$ (the blue area on processor $\pi_j$ is void in that case). This situation is depicted in Figure 2.20 where idle$^{j-1}_2 = \text{idlle}^{j-1}_3 = \text{idlle}^{j-1}_4$ and the blue area is void on processors $\pi_2$ and $\pi_3$. In this kind of situation, the job $J_i$ is executed (if not completed) upon a faster processor and the first instant at which at least $j$ processors idle remains unchanged after having scheduled the job $J_i$, i.e., idle$^j_i = \text{idlle}^{j-1}_j$.

**Case 2.** Otherwise, $J_i$ is dispatched to processor $\pi_j$ at instant idle$^{j-1}_j$ and keeps executing on $\pi_j$ as long as (i) no faster processors become idle or (ii) $J_i$ completes. In the first case, $J_i$ executes on $\pi_j$ until the next idle-instant idle$^{j-1}_{j+1}$, leading to the first sub-case idle$^j_j = \text{idlle}^{j-1}_{j+1}$.

168
In the second case, \( J_i \) executes on processor \( \pi_j \) but completes before time \( \text{idle}^{i-1}_j \). Thus, the idle-instant \( \text{idle}^{i}_j \) is the instant \( \text{idle}^{i-1}_j + 1 \) at which \( J_i \) was dispatched to \( \pi_j \) plus its remaining processing time on processor \( \pi_j \) at time \( \text{idle}^{i-1}_j \). Since \( \sum_{k=1}^{j} (\text{idle}^{i-1}_k - \text{idle}^{i-1}_{k-1}) \cdot s_k \) corresponds to the amount of work that \( J_i \) has executed in the interval of time \( [0, \text{idle}^{i-1}_j] \), its remaining processing time on processor \( \pi_j \) at time \( \text{idle}^{i-1}_j \) is given by \( \frac{c_i \sum_{k=1}^{j} (\text{idle}^{i-1}_k - \text{idle}^{i-1}_{k-1}) \cdot s_k}{s_j} \), leading to the second sub-case.

![Figure 2.20: Staircase defined by the idle\(^{i-1}_j\)](image-url)
Corollary 2.8 (From P. Meumeu Yomsi, V. Nelis and J. Goossens [30])

Let \( \pi \) denote any uniform multiprocessor platform composed of \( m \) processors. Let \( S \) be any global, FTP and work-conserving scheduler and let \( J = \{ J_1, J_2, \ldots, J_n \} \) be any set of \( n \) jobs of respective processing time \( c_1, c_2, \ldots, c_n \). Suppose that \( J \) is ordered by decreasing \( S \)-priority, i.e., \( J_1 > S J_2 > S \cdots > S J_n \). If \( J \) is scheduled by \( S \) upon \( \pi \), then the maximum idle-instant idle \( \text{idle}_k(J, \pi, \mathcal{P}) \) (\( \forall k \in [1, m] \)) is given by idle \( n \) computed as in Lemma 2.29.

Corollary 2.9

Let \( \pi \) denote any uniform multiprocessor platform composed of \( m \) processors. Let \( S \) be any global, FTP and work-conserving scheduler and let \( J = \{ J_1, J_2, \ldots, J_n \} \) be any set of \( n \) jobs of respective processing time \( c_1, c_2, \ldots, c_n \). Suppose that \( J \) is ordered by decreasing \( S \)-priority, i.e., \( J_1 > S J_2 > S \cdots > S J_n \). If \( J \) is scheduled by \( S \) upon \( \pi \), then the maximum makespan \( \text{ms}_{\text{unif}}(J, \pi) \) is given by idle \( n \) computed as in Lemma 2.29.

2.9.2 Determination of a validity test

From Corollary 2.9, a sufficient validity test for the protocol SM-MSO can therefore be formalized as follows.

Validity Test 2.6 (For SM-MSO, FTP schedulers and uniform platforms)

For any multimode real-time application \( \tau \) and any identical platform \( \pi \) composed of \( m \) processors, the protocol SM-MSO is valid provided that, for every mode \( M_i \),

\[
\text{idle}_i^{\text{ms}}(J_i^{\text{wc}}, \pi, S_i) \leq \min_{j \neq i} \left\{ \min_{k=1}^{n_j} \left\{ D_k^i(M_i) \right\} \right\}
\]

where idle \( n_i \) is computed as idle \( n_i \) in Lemma 2.29, considering the worst-case rem-job set \( J_i^{\text{wc}} \) composed of \( n_i \) jobs \( J_1, J_2, \ldots, J_n \) of respective processing time \( C_1, C_2, \ldots, C_n \), and such that \( J_i^{\text{wc}} \) is ordered by decreasing \( S_i \)-priority.

Similarly, the upper-bounds \( \text{idle}_k(J, \pi, \mathcal{P}) \) (where \( 1 \leq k \leq m \) and \( \mathcal{P} \) corresponds to the job priority assignment of the old-mode scheduler \( S_i \)) determined in Lemma 2.29 can be used at line 10 of the validity algorithm of AM-MSO (see Algorithm 4 page 99), as.
2.10 Adaptation to identical platforms of the upper-bound for uniform platforms

long as these upper-bounds are computed while assuming the worst-case rem-job set $F^w_c$ for the transitions from every mode $M_i$.

2.10 Adaptation to identical platforms of the upper-bound for uniform platforms

In this section, we investigate whether the upper-bounds on the idle-instants that we determined for uniform platforms can be used in the context of identical platforms. In Figure 2.11 (page 114), we depicted a chart of the different upper-bounds presented in this chapter, as well as the relation between each other. This figure is repeated in Figure 2.21.

With no loss of generality, we assume that the speed of every processor of every identical platform is equal to 1. That is, any identical platform can be modeled by the uniform platform model $\pi = [s_1, s_2, \ldots, s_m]$, where $s_i = 1 \forall i \in [1, m]$. Assuming that $\pi$ is any identical platform:

- Lemma 2.30 shows that the upper-bound $ms^{unif}_1(J, \pi)$ defined in Corollary 2.5 (page 137) does not compete with the upper-bound $ms^{ident}_1(J, \pi)$ defined in Corollary 2.2 (page 124). That is, we show that the upper-bound $ms^{unif}_1(J, \pi)$ is always more pessimistic than $ms^{ident}_1(J, \pi)$ if $\pi$ is an identical platform, i.e., for every set $J$ of jobs and every identical platform $\pi$:

$$ms^{unif}_1(J, \pi) \geq ms^{ident}_1(J, \pi)$$

This is the relation (a) depicted in Figure 2.21.

- Lemma 2.31 shows that the upper-bound $ms^{unif}_2(J, \pi)$ defined in Lemma 2.22 (page 160) is a generalization of $ms^{ident}_2(J, \pi)$. In other words, $ms^{unif}_2(J, \pi)$ provides an upper-bound on the makespan for any uniform platform $\pi$ but if $\pi$ is an identical platform then we have

$$ms^{unif}_2(J, \pi) = ms^{ident}_2(J, \pi)$$

- Lemma 2.32 rewrites the upper-bound $ms^{unif}_3(J, \pi)$ defined in Lemma 2.28 (page 165) for any identical multiprocessor platform $\pi$.
Figure 2.21: Chart representation of the contributions presented in this chapter, where the labels (a), (b), (c) and (d) illustrate the relation between the contributions: (a) the upper-bound $ms^\text{ident} (J, \pi)$ is more pessimistic than $ms^\text{unif} (J, \pi)$, (b) the upper-bound $ms^\text{unif} (J, \pi)$ is a generalization of $ms^\text{ident} (J, \pi)$, (c) we conjecture that the upper-bound $ms^\text{unif} (J, \pi)$ is more pessimistic than $ms^\text{ident} (J, \pi)$ and (d) the computation of $ms^\text{unif} (J, \pi, P)$ and $ms^\text{ident} (J, \pi, P)$ are very different for FTP schedulers, especially because of the difference in the definitions of weakly and strongly work-conserving schedulers.

**Lemma 2.30**

Let $\pi$ denote any identical multiprocessor platform composed of $m$ processors and let $J$ be any set of $n$ jobs such that $c_1 \leq c_2 \leq \cdots \leq c_n$. The upper-bound $ms^\text{ident} (J, \pi)$ defined in Corollary 2.2 is never larger than the upper-bound $ms^\text{unif} (J, \pi)$ defined in Corollary 2.5, i.e.,

$$ms^\text{ident} (J, \pi) \leq ms^\text{unif} (J, \pi)$$

**Proof**

From Corollary 2.5, it holds that
2.10 Adaptation to identical platforms of the upper-bound for uniform platforms

\[
\overline{m_s}^{\text{unif}}_1(J, \pi) \overset{\text{def}}{=} \frac{1}{s_m} \left( \sum_{i=1}^{n} c_i - \sum_{k=1}^{m-1} \text{idle}_k \cdot s_k \right)
\]

\[
= \frac{\sum_{i=1}^{n} c_i}{s_m} - \frac{\sum_{k=1}^{m-1} \left( \sum_{i=1}^{n-m+k} c_i \cdot s_k \right)}{s_m \cdot s(1)}
\]

By replacing every speed with 1, this yields

\[
\overline{m_s}^{\text{unif}}_1(J, \pi) = \sum_{i=1}^{n} c_i - \frac{\sum_{k=1}^{m-1} \left( \sum_{i=1}^{n-m+k} c_i \cdot 1 \right)}{1 \cdot m}
\]

(2.58)

As shown in Figure 2.22, the following equality holds

\[
\sum_{k=1}^{m-1} \left( \sum_{i=1}^{n-m+k} c_i \right) = (m - 1) \sum_{i=1}^{n} c_i + \sum_{i=1}^{m-1} (m - i) \cdot c_{n-m+i}
\]

and Equality 2.58 can therefore be rewritten as

\[
\overline{m_s}^{\text{unif}}_1(J, \pi) = \sum_{i=1}^{n} c_i - \frac{(m - 1) \cdot \sum_{i=1}^{n} c_i + \sum_{i=1}^{m-1} (m - i) \cdot c_{n-m+i}}{m}
\]

leading to

\[
\overline{m_s}^{\text{unif}}_1(J, \pi) = \sum_{i=1}^{n} c_i - \frac{m \cdot \sum_{i=1}^{n-m} c_i}{m} + \frac{\sum_{i=1}^{n-m} c_i}{m} - \frac{m \cdot \sum_{i=1}^{m-1} c_{n-m+i}}{m} + \frac{\sum_{i=1}^{m-1} i \cdot c_{n-m+i}}{m}
\]

Since \( \sum_{i=1}^{n} c_i - \sum_{i=1}^{n-m} c_i - \sum_{i=1}^{m-1} c_{n-m+i} = \sum_{i=1}^{n} c_i - \sum_{i=1}^{n-1} c_i = c_n \), we get

\[
\overline{m_s}^{\text{unif}}_1(J, \pi) \geq c_n + \frac{\sum_{i=1}^{n-m} c_i}{m} + \frac{\sum_{i=1}^{m-1} i \cdot c_{n-m+i}}{m}
\]

\[
\geq c_n + \frac{\sum_{i=1}^{n-m} c_i}{m} + \frac{\sum_{i=1}^{m-1} c_{n-m+i}}{m}
\]

\[
\geq c_n + \frac{\sum_{i=1}^{n} c_i}{m}
\]

\[
\geq \overline{m_s}^{\text{ident}}_1(J, \pi) \overset{\text{def}}{=} \text{defined as in Expression 2.13 on page 124}
\]

173
(a) By reading the matrix line by line (from up to bottom), one can easily see that this illustration represents the term $\sum_{i=1}^{m-1} \sum_{k=1}^{n-m+k} c_i$.

$$\sum_{i=1}^{m-1} c_i \text{ with } i = 1 \rightarrow n - m + k$$

$$(k = 1) \quad c_1 + c_2 + c_3 + \ldots + c_{n-m+1}$$

$$(k = 2) \quad + c_1 + c_2 + c_3 + \ldots + c_{n-m+1} + c_{n-m+2}$$

$$\vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots$$

$$(k = m-2) \quad + c_1 + c_2 + c_3 + \ldots + c_{n-m+1} + c_{n-m+2} + \ldots + c_{n-2}$$

$$(k = m-1) \quad + c_1 + c_2 + c_3 + \ldots + c_{n-m+1} + c_{n-m+2} + \ldots + c_{n-2} + c_{n-1}$$

(b) By reading the matrix column by column (from left to right), this matrix can be rewritten as $(m - 1) \cdot \sum_{i=1}^{n-m} c_i + \sum_{i=1}^{m-1} (m-i) \cdot c_{n-m+i}$.

$$(m - 1) \cdot \sum_{i=1}^{n-m} c_i + \sum_{i=1}^{m-1} (m-i) \cdot c_{n-m+i}$$

$$(k = 1) \quad c_1 \cdot (m-1)$$

$$(k = 2) \quad c_2 \cdot (m-1)$$

$$(k = m-2) \quad c_{n-m+1} \cdot (m-1)$$

$$(k = m-1) \quad c_{n-1} \cdot (m-1)$$

$$\vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots$$

$$(k = m-2) \quad c_{n-m+1} \cdot (m-2)$$

$$(k = m-1) \quad c_{n-1} \cdot (m-2)$$

$$(m-1) \cdot c_{n-1}$$

Figure 2.22: Illustration of the equality $\sum_{k=1}^{m-1} \sum_{i=1}^{n-m+k} c_i = (m - 1) \cdot \sum_{i=1}^{n-m} c_i + \sum_{i=1}^{m-1} (m-i) \cdot c_{n-m+i}$.
2.10 Adaptation to identical platforms of the upper-bound for uniform platforms

As a result, if the considered platform $\pi$ is identical then it is not interesting to use the upper-bound $\ms_{1}^{\text{unif}}(J, \pi)$ in the validity test of SM-MSO since the upper-bound $\ms_{1}^{\text{ident}}(J, \pi)$ always outperforms (or is equivalent to) $\ms_{1}^{\text{unif}}(J, \pi)$.

Lemma 2.31

Let $\pi$ denote any identical multiprocessor platform composed of $m$ processors and let $J$ be any set of $n$ jobs such that $c_1 \leq c_2 \leq \cdots \leq c_n$. The upper-bound $\ms_{2}^{\text{unif}}(J, \pi)$ defined in Lemma 2.22 is a generalization to uniform platform of the upper-bound $\ms_{\text{ident}}^{\text{ident}}(J, \pi)$ defined in Corollary 2.2.

Proof

From Lemma 2.22, it holds that

$$\ms_{2}^{\text{unif}}(J, \pi) \overset{\text{def}}{=} \frac{1}{s_m} \cdot \sum_{i=1}^{n} \left( c_i + \frac{\sum_{j=1}^{i-1} c_j}{s(1)} \right) \cdot K_{n-i}$$

where $K_{n-i}$ is such that

$$K_{n-i} \overset{\text{def}}{=} \begin{cases} 1 & \text{if } n-i = 0 \\ \left(1 - \frac{s_1}{s_m}\right)^{n-i} & \text{otherwise} \end{cases}$$

Thus, replacing the speeds $s_1, s_2, \ldots, s_m$ with 1 yields

$$\ms_{2}^{\text{unif}}(J, \pi) = \frac{1}{1} \cdot \sum_{i=1}^{n} \left( c_i + 1 \cdot \frac{\sum_{j=1}^{i-1} c_j}{m} \right) \cdot K_{n-i}$$

Since $s_1 = s_m$, all the factors $K_{n-i}$ are equal to 0, except for $i = n$ since $K_0 \overset{\text{def}}{=} 1$. Thereby, the above equality yields

$$\ms_{2}^{\text{unif}}(J, \pi) = c_n + \frac{1}{m} \cdot \sum_{j=1}^{n-1} c_j$$

which corresponds to the upper-bound $\ms_{2}^{\text{ident}}(J, \pi)$ defined by Expression 2.13 (page 124) for FJP schedulers and identical platforms. The lemma follows.
Lemma 2.32

Let \( \pi \) denote any identical multiprocessor platform composed of \( m \) processors and let \( J \) be any set of \( n \) jobs such that \( c_1 \leq c_2 \leq \cdots \leq c_n \). The upper-bound \( m_{\text{unif}}^{3}(J, \pi) \) defined in Lemma 2.28 on page 165 yields

\[
m_{\text{unif}}^{3}(J, \pi) = \sum_{\ell=1}^{n} \left( c_{\ell} + \frac{1}{m^2} \cdot \sum_{j=1}^{\ell-1} c_{j} \right) \cdot \left( 1 - \frac{1}{m} \right)^{n-\ell} \tag{2.59}
\]

Proof

From Lemma 2.28, it holds that

\[
m_{\text{unif}}^{3}(J, \pi) \overset{\text{def}}{=} \frac{1}{s_{m}} \cdot \sum_{\ell=1}^{n} \left( c_{\ell} + \frac{s_{x} \cdot s_{m}}{s(1) \cdot \sum_{j=1}^{\ell} s_{j}} \cdot \sum_{j=1}^{\ell-1} c_{j} \right) \cdot H_{n-\ell}
\]

where \( x = m \) (i.e., the critical speed \( s_{x} \) is the speed \( s_{m} \)) and since \( s_{x} \neq \sum_{i=1}^{x} s_{i} \) then \( H_{n-\ell} \) is such that

\[
H_{n-\ell} \overset{\text{def}}{=} \left( 1 - \frac{s_{x}}{\sum_{i=1}^{\ell} s_{i}} \right)^{n-\ell}
\]

Thus, replacing the speeds \( s_{1}, s_{2}, \ldots, s_{m} \) and \( s_{x} \) with 1 yields

\[
m_{\text{unif}}^{3}(J, \pi) = \frac{1}{1} \cdot \sum_{\ell=1}^{n} \left( c_{\ell} + \frac{1}{m^2} \cdot \sum_{j=1}^{\ell-1} c_{j} \right) \cdot H_{n-\ell}
\]

\[
= \sum_{\ell=1}^{n} \left( c_{\ell} + \frac{1}{m^2} \cdot \sum_{j=1}^{\ell-1} c_{j} \right) \cdot \left( 1 - \frac{1}{m} \right)^{n-\ell}
\]

We conjecture that, for every set \( J \) of jobs and every identical multiprocessor platform \( \pi \), the upper-bound \( m_{\text{unif}}^{3}(J, \pi) \) defined by Expression 2.59 is more pessimistic than the upper-bound \( m_{\text{ident}}(J, \pi) \) defined in Corollary 2.2 on page 124, i.e.,

\[
m_{\text{ident}}(J, \pi) \leq m_{\text{unif}}^{3}(J, \pi)
\]
2.11 Accuracy of the proposed upper-bounds

This section provides an analysis of the accuracy of the upper-bounds on the makespan that we determined in this chapter, i.e., the four upper-bounds \( \text{ms}^{\text{ident}}(J, \pi) \), \( \text{ms}^{\text{unif}}_1(J, \pi) \), \( \text{ms}^{\text{unif}}_2(J, \pi) \) and \( \text{ms}^{\text{unif}}_3(J, \pi) \) determined in Expressions 2.13 (page 124), 2.20 (page 137), 2.48 (page 161) and 2.56 (page 166), respectively. First, we prove in Lemma 2.33 below that the upper-bound \( \text{ms}^{\text{ident}}(J, \pi) \) is 2-competitive, with the interpretation that the value returned by \( \text{ms}^{\text{ident}}(J, \pi) \) is at most twice the actual maximum makespan. Then, we identify in the following three lemmas 2.35, 2.36 and 2.37 a factor \( \alpha_i(J, \pi) \) for each upper-bound \( \text{ms}^{\text{unif}}_i(J, \pi), i = 1, 2, 3 \), such that the value returned by \( \text{ms}^{\text{unif}}_i(J, \pi) \) is at most \( \alpha_i(J, \pi) \) times greater than the actual maximum makespan. Unfortunately, for the three expressions \( \text{ms}^{\text{unif}}_i(J, \pi) \) we failed in expressing a competitive factor independent of the jobs and platform characteristics.

For each upper-bound, its associated competitive factor is determined while taking the following two points into consideration.

1. During any mode transition from any mode \( M^i \) to any other mode \( M^j \), the minimum makespan that could be produced is always 0 since it can always be the case that no old-mode task has an active job at time \( t_{\text{MCR}(j)} \), i.e., when the mode change is requested. For instance in Figure 2.3 page 86, the makespan would be zero if the MCR\((j)\) was released at time 110. However, in order to guarantee that our approaches always provide upper-bounds on the makespan, we have to consider the worst-case scenario, i.e., the scenario in which every old-mode task releases a job exactly at time \( t_{\text{MCR}(j)} \) and all these jobs executes for their WCET during the transition.

2. Assuming this worst-case scenario (that we named “worst-case rem-job set” in our study), we proposed in Corollaries 2.3 and 2.8, a method for FTP schedulers that provides the exact idle-instants (and thus an exact value of the makespan), considering identical and uniform platforms, respectively. That is, the accuracy of these two methods is maximum and we only focus here on the accuracy of the upper-bounds on the makespan for FJP schedulers.
Lemma 2.33

For any set $J$ of jobs ordered by non-decreasing job processing time and for any identical multiprocessor platform $\pi$ composed of $m$ processors, the upper-bound $\overline{\text{ms}}^{\text{ident}}(J, \pi)$ is 2-competitive.

**Proof**

Recall that, from Expression 2.13,

\[
\overline{\text{ms}}^{\text{ident}}(J, \pi) \overset{\text{def}}{=} \begin{cases} 
    c_n & \text{if } (n \leq m) \\
    \sum_{i=1}^{n-1} c_i + c_n & \text{otherwise}
\end{cases}
\]

Let $\text{ms}(J, m)$ denote the exact makespan for the set $J$ of jobs and the $m$ identical processors. Since we do not have any mathematical expression for determining this exact makespan $\text{ms}(J, m)$, our analysis is performed while considering a lower-bound $\underline{\text{ms}}^{\text{ident}}(J, m)$ on the makespan rather than its exact value, i.e., $\alpha$ is determined in such a manner that

\[
\frac{\overline{\text{ms}}^{\text{ident}}(J, \pi)}{\underline{\text{ms}}^{\text{ident}}(J, m)} \leq \alpha
\]

where

\[
\underline{\text{ms}}^{\text{ident}}(J, m) \overset{\text{def}}{=} \begin{cases} 
    c_n & \text{if } n \leq m \\
    \max\left(c_n, \frac{\sum_{i=1}^{n-1} c_i}{m}\right) & \text{if } n > m
\end{cases}
\]

The case where $n \leq m$ obviously leads to $\alpha = 1$ since both the upper-bound $\overline{\text{ms}}^{\text{ident}}(J, \pi)$ and the lower-bound $\underline{\text{ms}}^{\text{ident}}(J, \pi)$ return a makespan of $c_n$. Otherwise (if $n > m$) the “max” operator in the definition of $\underline{\text{ms}}^{\text{ident}}(J, m)$ leads to two cases for determining $\alpha$.

**Case 1.** If $c_n \geq \frac{\sum_{i=1}^{n-1} c_i}{m}$ then we get

\[
\frac{\overline{\text{ms}}^{\text{ident}}(J, \pi)}{\text{ms}(J, m)} \leq \frac{\overline{\text{ms}}^{\text{ident}}(J, \pi)}{\underline{\text{ms}}^{\text{ident}}(J, m)} \leq \frac{\sum_{i=1}^{n-1} c_i + c_n}{c_n} \leq \frac{\sum_{i=1}^{n-1} c_i + c_n}{c_n}
\]
2.11 Accuracy of the proposed upper-bounds

and since in this case we have $c_n \geq \sum_{i=1}^{n} c_i$, it holds that

$$\frac{\text{ms}^{\text{ident}}(J, \pi)}{\text{ms}(J, m)} \leq \frac{c_n + c_n}{c_n} \leq 2$$

**Case 2.** If $c_n < \frac{\sum_{i=1}^{n} c_i}{m}$ then

$$\frac{\text{ms}^{\text{ident}}(J, \pi)}{\text{ms}(J, m)} \leq \frac{\text{ms}^{\text{ident}}(J, \pi)}{\text{ms}^{\text{ident}}(J, m)} \leq \frac{\sum_{i=1}^{n-1} c_i}{m} + \frac{c_n}{m}$$

and since in this case we have $c_n < \frac{\sum_{i=1}^{n} c_i}{m}$, it holds that

$$\frac{\text{ms}^{\text{ident}}(J, \pi)}{\text{ms}(J, m)} \leq \frac{\sum_{i=1}^{n-1} c_i}{m} + \frac{\sum_{i=1}^{n} c_i}{m} \leq 2$$

It holds from the above lemma that, for any set $J$ of jobs and any identical platform composed of $m$ processors, the upper-bound on the maximum makespan provided by $\text{ms}^{\text{ident}}(J, \pi)$ is at most $\alpha = 2$ times the exact value of the maximum makespan. Additionally, we show in the following Lemma 2.34 that the upper-bounds $\text{idle}_k(J, \pi)$ ($\forall k \in [1, m]$) defined for identical platforms on page 121 are exact in some particular cases (thus it also holds for $\text{ms}^{\text{ident}}(J, \pi)$ since $\text{ms}^{\text{ident}}(J, \pi) \overset{\text{def}}{=} \text{idle}_m(J, \pi)$).

<table>
<thead>
<tr>
<th>$c_1$</th>
<th>$c_2$</th>
<th>$c_3$</th>
<th>$c_4$</th>
<th>$c_5$</th>
<th>$c_6$</th>
<th>$c_7$</th>
<th>$c_8$</th>
<th>$c_9$</th>
<th>$c_{10}$</th>
<th>$c_{11}$</th>
<th>$c_{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

**Table 2.1:** Processing times of the 12 jobs in $J$.  

179
Figure 2.23: Illustration of different schedules proving that the upper-bounds provided by Expression 2.12 can be reached.
2.11 Accuracy of the proposed upper-bounds

(a) Schedule of the jobs $J_1, J_2, J_3$ where $J_1 > J_2 > J_3$.

(b) Illustration of how the maximum makespan is estimated by Expression 2.13.

Figure 2.24: Illustration of a schedule in which the makespan is over-approximated.
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

Lemma 2.34

For any identical multiprocessor platform $\pi$ composed of $m$ processors, there exists some sets $J$ of jobs for which the upper-bounds $\text{idle}_k(J, \pi)$ ($\forall k \in [1, m]$) defined by Expression 2.11 (page 121) are exact. Therefore, this also holds for the upper-bound $\text{ms}^{\text{ident}}(J, \pi)$ on the makespan since $\text{ms}^{\text{ident}}(J, \pi) = \text{idle}_m(J, \pi)$.

Proof

Let $J$ be the set of 12 jobs $J_1, J_2, \ldots, J_{12}$ with processing times given in Table 2.1 (assuming that jobs are indexed such that $c_i \leq c_{i+1} \forall i$) and suppose that $J$ is scheduled upon 3 identical processors. According to Expression 2.13, since $n = 12$ the makespan is bounded from above by

$$\frac{\sum_{j=1}^{n} c_j + (m-1) \cdot c_n}{m} = \frac{45 + (3-1) \cdot 12}{3} = 23$$

and we can see in Figure 2.23a that this upper-bound can actually be reached assuming the priority assignment:

$$J_7 > J_11 > J_10 > J_1 > J_2 > J_9 > J_8 > J_3 > J_5 > J_4 > J_6 > J_{12}$$

Similarly, Expression 2.12 upper-bounds the idle-instants idle$_2$ and idle$_1$ by

$$\frac{\sum_{j=1}^{n} c_j + (2-1) \cdot c_n}{3} = \frac{45 + 9}{3} = 18$$ and $$\frac{\sum_{j=1}^{n} c_j}{3} = 15$$, respectively. As we can see in Figure 2.23b and 2.23c, these upper-bounds can also be reached, while considering the priority assignments

$$J_{10} > J_9 > J_1 > J_2 > J_3 > J_4 > J_5 > J_6 > J_{12} > J_7 > J_8 > J_{11}$$

and

$$J_7 > J_9 > J_{10} > J_{12} > J_{11} > J_6 > J_1 > J_2 > J_3 > J_4 > J_5 > J_6$$

respectively. However, we worth notice that in most cases, these upper-bounds idle$_k$ over-approximate the idle-instants. For instance, consider the set $J$ of 3 jobs $J_1, J_2, J_3$ of respective processing time 4, 10, 14 and suppose that $J$ is scheduled upon 2 identical processors. The maximum makespan is 18 (reached with the priority assignment: $J_1 > J_2 > J_3$). However, according to Expression 2.13, the maximum makespan is bounded from above by $\frac{4+10}{2} + 14 = 21$. Figure 2.24a depicts the schedule of $J$ assuming $J_1 > J_2 > J_3$ and Figure 2.24b illustrates the computation of the maximum makespan by Expression 2.13.
2.11 Accuracy of the proposed upper-bounds

Lemma 2.35

For any set $J$ of jobs ordered by non-decreasing job processing time and any uniform platform $\pi = [s_1, s_2, \ldots, s_m]$ with $s_i \leq s_{i+1}$ for all $i$, $\bar{m}_1^\text{unif}(J, \pi)$ is $\alpha_1(\pi)$-competitive, where

$$\alpha_1(\pi) \overset{\text{def}}{=} \frac{s(1)}{s_m}$$

Proof

Recall that, from Expression 2.20,

$$\bar{m}_1^\text{unif}(J, \pi) \overset{\text{def}}{=} \frac{\sum_{i=1}^{n} c_i}{s_m} - \frac{\sum_{k=1}^{m-1} \left( \sum_{i=1}^{n-m+k} c_i \cdot s_k \right)}{s_m \cdot s(1)}$$

Let $m(J, \pi)$ denote the exact makespan for any given set $J$ of jobs and any uniform platform $\pi$. Since we do not have any mathematical expression for determining this exact makespan $m(J, \pi)$, our analysis of $\alpha$ is performed while considering a lower-bound $\bar{m}(J, \pi)$ on the makespan rather than its exact value, i.e., $\alpha$ is determined in such a manner that

$$\frac{\bar{m}_1^\text{unif}(J, \pi)}{m(J, \pi)} \leq \alpha$$

Obviously, we know that $m(J, \pi) \geq \frac{\sum_{i=1}^{n} c_i}{s(1)}$ and this implies that $\bar{m}(J, \pi) = \frac{\sum_{i=1}^{n} c_i}{s(1)}$ is a lower-bound on the makespan. This yields

$$\frac{\bar{m}_1^\text{unif}(J, \pi)}{m(J, \pi)} \leq \frac{\bar{m}_1^\text{unif}(J, \pi)}{\bar{m}(J, \pi)}$$

and thus,

$$\frac{\bar{m}_1^\text{unif}(J, \pi)}{m(J, \pi)} \leq \frac{\sum_{i=1}^{n} c_i}{s_m} - \frac{\sum_{k=1}^{m-1} \left( \sum_{i=1}^{n-m+k} c_i \cdot s_k \right)}{s_m \cdot s(1)} \leq \left( \frac{\sum_{i=1}^{n} c_i}{s_m} - \frac{\sum_{k=1}^{m-1} \left( \sum_{i=1}^{n-m+k} c_i \cdot s_k \right)}{s_m \cdot s(1)} \right) \frac{s(1)}{\sum_{i=1}^{n} c_i} \quad (2.60)$$
\[ \sum_{i=1}^{n} c_i \cdot s_i \leq \left( \frac{\sum_{i=1}^{n} c_i}{s_m} \right) \cdot \frac{s(1)}{s_m} \leq \frac{s(1)}{s_m} \]  
(2.61)

Notice the important loss of accuracy that this inequality underwent when we ignored the term 
\[ - \sum_{k=1}^{m-1} \left( \sum_{i=1}^{n-m+k} c_i \cdot s_k \right) \cdot s_m \cdot s(1) \]  
while passing from Inequality 2.60 to Inequality 2.61.

Lemma 2.36

For any set \( J \) of jobs ordered by non-decreasing job processing time and any uniform platform \( \pi = [s_1, s_2, \ldots, s_m] \) with \( s_i \leq s_{i+1} \) \( \forall i \), \( \overline{ms}_2^{unif}(J, \pi) \) is \( \alpha_2(J, \pi) \)-competitive, where 
\[ \alpha_2(J, \pi) \overset{\text{def}}{=} \frac{s(1) + n \cdot s_1}{s_m} \]

Proof

Recall that, from Expression 2.48,
\[ \overline{ms}_2^{unif}(J, \pi) \overset{\text{def}}{=} \frac{1}{s_m} \cdot \sum_{i=1}^{n} \left( c_i + s_1 \cdot \frac{\sum_{j=1}^{i-1} c_j}{s(1)} \right) \cdot K_{n-i} \]
where \( K_j \) is such that \( \forall j \),
\[ K_j \overset{\text{def}}{=} \begin{cases} 1 & \text{if } s_1 = s_m \text{ and } j = 0 \\ \left(1 - \frac{s_1}{s_m}\right)^j & \text{otherwise} \end{cases} \]
As for the previous analyses of \( \alpha \), we use the lower-bound \( \overline{ms}(J, \pi) = \frac{\sum_{i=1}^{n} c_i}{s(1)} \) instead of the exact but unknown value \( ms(J, \pi) \) of the makespan. This yields
\[ \frac{\overline{ms}_2^{unif}(J, \pi)}{ms(J, \pi)} \leq \frac{\overline{ms}(J, \pi)}{\overline{ms}(J, \pi)} \]
and thus,

\[
\frac{\overline{ms}_{2}^{\text{unif}}(J, \pi)}{ms(J, \pi)} \leq \frac{1}{s_m} \cdot \frac{\sum_{i=1}^{n} (c_i + s_1 \cdot \frac{\sum_{j=1}^{i-1} c_j}{s(1)}) \cdot K_{n-i}}{\sum_{i=1}^{n} c_i / s(1)}
\]

Since \( K_j \leq 1 \ \forall j \), we have

\[
\frac{\overline{ms}_{2}^{\text{unif}}(J, \pi)}{ms(J, \pi)} \leq \frac{1}{s_m} \cdot \frac{\sum_{i=1}^{n} (c_i + s_1 \cdot \frac{\sum_{j=1}^{i-1} c_j}{s(1)})}{\sum_{i=1}^{n} c_i / s(1)} \cdot \frac{s(1)}{s_m \cdot \sum_{i=1}^{n} c_i}
\]

and since \( \sum_{j=1}^{i-1} c_j < \sum_{j=1}^{n} c_j \) \( \forall i, 1 \leq i \leq n \), it holds that

\[
\frac{\overline{ms}_{2}^{\text{unif}}(J, \pi)}{ms(J, \pi)} < \sum_{i=1}^{n} \left( \frac{c_i + s_1 \cdot \sum_{j=1}^{i-1} c_j}{s(1)} \right) \cdot \frac{s(1)}{s_m \cdot \sum_{i=1}^{n} c_i}
\]

\[
< \frac{\sum_{i=1}^{n} c_i \cdot s(1)}{s_m \cdot \sum_{i=1}^{n} c_i} + \frac{n \cdot s_1 \cdot \sum_{j=1}^{n} c_j \cdot s(1)}{s(1) \cdot s_m \cdot \sum_{i=1}^{n} c_i}
\]

\[
< \frac{s(1)}{s_m} + n \cdot \frac{s_1}{s_m}
\]

Lemma 2.37

For any set \( J \) of jobs ordered by non-decreasing job processing time and any uniform platform \( \pi = [s_1, s_2, \ldots, s_m] \) with \( s_i \leq s_{i+1} \ \forall i \), \( \overline{ms}_{3}^{\text{unif}}(J, \pi) \) is \( \alpha_3(J, \pi) \)-competitive, where

\[
\alpha_3(J, \pi) \overset{\text{def}}{=} \frac{s(1)}{s_m} + n \cdot \frac{s_x}{\sum_{j=1}^{n} s_j}
\]

and \( x = \arg\min_{i \in [1,m]} \left\{ \frac{s_i}{\sum_{j=1}^{n} s_j} \right\} \).

Proof

Recall that, from Expression 2.56,
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

\[ ms_{\text{unif}}^3(J, \pi) \overset{\text{def}}{=} \frac{1}{s_m} \sum_{\ell=1}^n \left( c_{\ell} + \frac{s_x \cdot s_m}{s(1) \sum_{j=1}^{\ell-1} s_j} \right) \cdot H_{n-\ell} \]

where \( H_j \) is such that \( \forall j, \)

\[ H_j \overset{\text{def}}{=} \begin{cases} 1 & \text{if } s_x = \sum_{i=1}^x s_i \text{ and } j = 0 \\ \left(1 - \frac{s_x}{\sum_{i=1}^x s_i}\right)^j & \text{otherwise} \end{cases} \]

As for the previous analyses of \( \alpha \), we use the lower-bound \( \overline{ms}(J, \pi) = \frac{\sum_{\ell=1}^n c_{\ell}}{s(1)} \) instead of the exact but unknown value \( ms(J, \pi) \) of the makespan. This yields

\[ \frac{ms_{\text{unif}}^3(J, \pi)}{ms(J, \pi)} \leq \frac{\overline{ms}(J, \pi)}{ms(J, \pi)} \leq \frac{1}{s_m} \cdot \sum_{\ell=1}^n \left( c_{\ell} + \frac{s_x \cdot s_m}{s(1) \sum_{j=1}^{\ell-1} s_j} \cdot \sum_{j=1}^{\ell-1} c_j \right) \cdot \frac{H_{n-\ell}}{s(1)} H_{n-\ell} \]

Since \( H_j \leq 1 \forall j \), we have

\[ \frac{ms_{\text{unif}}^3(J, \pi)}{ms(J, \pi)} \leq \frac{1}{s_m} \cdot \sum_{\ell=1}^n \left( c_{\ell} + \frac{s_x \cdot s_m}{s(1) \sum_{j=1}^{\ell-1} s_j} \cdot \sum_{j=1}^{\ell-1} c_j \right) \cdot \frac{1}{s(1)} \sum_{i=1}^n c_i \]

and since \( \sum_{j=1}^{\ell-1} c_j < \sum_{j=1}^n c_j \forall \ell, 1 \leq i \leq n \), it holds that

\[ \frac{ms_{\text{unif}}^3(J, \pi)}{ms(J, \pi)} < \sum_{\ell=1}^n \left( c_{\ell} + \frac{s_x \cdot s_m}{s(1) \sum_{j=1}^n s_j} \cdot \sum_{j=1}^n c_j \right) \cdot \frac{1}{s_m} \cdot \sum_{i=1}^n c_i \]

leading to

\[ \frac{ms_{\text{unif}}^3(J, \pi)}{ms(J, \pi)} < \frac{\sum_{\ell=1}^n c_{\ell} \cdot s(1) + s_x \cdot s_m \cdot s(1) \cdot n \cdot \sum_{i=1}^n c_i}{s_m \cdot \sum_{i=1}^n c_i} + \frac{s_x \cdot s_m \cdot \sum_{j=1}^x s_j \cdot s_m \cdot \sum_{i=1}^n c_i}{s(1) \cdot \sum_{j=1}^x s_j \cdot s_m \cdot \sum_{i=1}^n c_i} \]

\[ < \frac{s(1)}{s_m} + n \cdot \frac{s_x}{\sum_{j=1}^x s_j} \]
2.12 Simulation results

As we can see, the competitive factors of our two upper-bounds $\overline{ms}_2^{\text{unif}}(J, \pi)$ and $\overline{ms}_3^{\text{unif}}(J, \pi)$ as computed above are larger than the competitive factor of $\overline{ms}_1^{\text{unif}}(J, \pi)$. However, these competitive factors are estimated by drastically reducing the exact expressions of $\overline{ms}_2^{\text{unif}}(J, \pi)$ and $\overline{ms}_3^{\text{unif}}(J, \pi)$, and we show in the case study presented in the next section that these two upper-bounds can strongly outperform $\overline{ms}_1^{\text{unif}}(J, \pi)$ in practice. Table 2.2 outlines the competitive factor of the upper-bounds presented in this chapter.

<table>
<thead>
<tr>
<th></th>
<th>$\overline{ms}^{\text{ident}}(J, \pi)$</th>
<th>$\overline{ms}_1^{\text{unif}}(J, \pi)$</th>
<th>$\overline{ms}_2^{\text{unif}}(J, \pi)$</th>
<th>$\overline{ms}_3^{\text{unif}}(J, \pi)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Competitive factor</td>
<td>2</td>
<td>$\frac{s(1)}{s_m}$</td>
<td>$\frac{s(1)+ns_1}{s_m}$</td>
<td>$\frac{s(1)}{s_m} + n \cdot \frac{s_f}{\sum_{j=1}^{s_f} j}$</td>
</tr>
</tbody>
</table>

Table 2.2: Competitive factors of the presented upper-bounds on the maximum makespan.

2.12 Simulation results

In this section, we report on the results of simulations conducted using the theoretical results presented for FJP schedulers. Indeed, the upper-bounds presented for FTP schedulers are exact if every job executes for its WCET and simulations are therefore useless for such schedulers. The simulations presented here are performed in order to analyze the precision of the four upper-bounds $\overline{ms}^{\text{ident}}(J, \pi), \overline{ms}_1^{\text{unif}}(J, \pi), \overline{ms}_2^{\text{unif}}(J, \pi)$ and $\overline{ms}_3^{\text{unif}}(J, \pi)$ for a given set of jobs $J$ scheduled upon a given platform $\pi$. During all our simulations, we consider a single set $J$ of jobs for which the exact processing times are given in Table 2.3. We will explain below where these parameters are drawn from and why we consider only a single set of jobs rather than generating numerous job sets.

<table>
<thead>
<tr>
<th>$c_1$</th>
<th>$c_2$</th>
<th>$c_3$</th>
<th>$c_4$</th>
<th>$c_5$</th>
<th>$c_6$</th>
<th>$c_7$</th>
<th>$c_8$</th>
<th>$c_9$</th>
<th>$c_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3896</td>
<td>3964</td>
<td>878</td>
<td>1378</td>
<td>2228</td>
<td>3612</td>
<td>1230</td>
<td>1232</td>
<td>1668</td>
<td>4672</td>
</tr>
</tbody>
</table>

Table 2.3: Processing times of the 10 jobs in $J$.

For experimental purposes, let us introduce the parameter $\lambda_{\pi}$ defined in [13] for any
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

\( m \)-processor uniform platform \( \pi = [s_1, s_2, \ldots, s_m] \),

\[
\lambda_\pi \overset{\text{def}}{=} \max_{j=1}^m \left\{ \frac{\sum_{k=1}^{j-1} s_k}{s_j} \right\}
\]

Informally, this parameter \( \lambda_\pi \) measures the “degree” by which \( \pi \) differs from an identical multiprocessor platform, i.e., its “degree of heterogeneity”. For any identical platform composed of \( m \) processors, it holds that \( s_1 = s_2 = \cdots = s_m \) and thus, \( \lambda_\pi \overset{\text{def}}{=} \max_{j=1}^m \left\{ \frac{\sum_{k=1}^{j-1} s_k}{s_j} \right\} \) is maximum for \( j = m \), leading to \( \lambda_\pi \overset{\text{def}}{=} \frac{\sum_{k=1}^{m-1} s_k}{s_m} = m - 1 \). More homogeneous the platform \( \pi = [s_1, s_2, \ldots, s_m] \) is, closer to \((m - 1)\) is its corresponding \( \lambda_\pi \). For instance, the uniform platform \( \pi = [1, 500, 1000] \) has a corresponding \( \lambda_\pi = \frac{501}{1000} \approx 0.5 \) whereas \( \lambda_\pi = \frac{501}{600} = 0.835 \) for the uniform platform \( \pi = [1, 500, 600] \) and \( \lambda_\pi = \frac{1000}{600} \approx 1.67 \) for the platform \( \pi = [500, 500, 600] \). In short, \( \lambda_\pi = (m - 1) \) if \( \pi \) is comprised of \( m \) identical processors and becomes progressively smaller as the speeds of the processors differ from each other by greater amounts.

The platform \( \pi \) considered in our simulations is composed of \( m = 4 \) processors for which we make their computing speed varying within [1, 101] with an increment of 10. More precisely, we consider all possible combinations of the processors speeds in the range [1, 101] with an increment of 10, i.e., the first simulation is performed considering \( \pi = [1, 1, 1, 1] \), then the second simulation considers \( \pi = [1, 1, 1, 11] \), the third one considers \( \pi = [1, 1, 1, 21] \), and so on until reaching the speed assignment \( \pi = [101, 101, 101, 101] \). For every speed assignment, we determine the corresponding parameter \( \lambda_\pi \) as well as the exact value \( \max_{MK(J, \pi)} \) of the maximum makespan. This exact maximum makespan \( \max_{MK(J, \pi)} \) is determined by building the schedule of \( J \) upon \( \pi \) for every job priority assignment and by recording the maximum resulting makespan. This is a highly computational-intensive operation that requires the exhaustive enumeration of every possible job priority assignment and this is the reason why we consider only a single set \( J \) of jobs in our simulations. Indeed, according to this approach, our simulation considers 11 different speeds for each processor, leading to a total of \( 11^m = 11^4 = 14,641 \) different platforms \( \pi \). For each platform \( \pi \), the computation of the exact makespan requires to generate the schedules derived from every job priority assignment. Since there are 10 jobs, the number of considered priority assignments is \( 10! = 3,628,800 \). Multiplied by the number of platforms, this leads to 53,129,260,800 operations. Our simulations were performed on HYDRA [21], the Scientific Computer Configuration at the VUB/ULB Computing Centre, where we fully distributed the com-
2.12 Simulation results

Computations among 15 processors AMD Opteron dual-core @ 2.8GHz. Distributing the computations allowed us to complete the simulation in about 2 hours but unfortunately, the amount of computations grows exponentially with the number of processors and in a factorial manner with the number of jobs. For instance, considering 13 jobs would result in $91,169,811,532,800$ operations, 14 jobs to approximately $20 \cdot 10^{15}$ operations, resulting in a computation time of about 82 years. The processing times of the jobs have been drawn from [10] where the authors present realistic parameters that concern the avionic domain. But since the number of operations of our algorithm is strongly restricted by the number of jobs, we randomly selected 10 WCETs from these parameters.

For each speed assignment of the platform we computed the error $E_{\text{unif}}^1(J, \pi)$ corresponding to the difference (in percent) between $\overline{ms}_{\text{unif}}^1(J, \pi)$ and maxMK$(J, \pi)$. Formally,

$$E_{\text{unif}}^1(J, \pi) \overset{\text{def}}{=} \frac{\overline{ms}_{\text{unif}}^1(J, \pi) - \text{maxMK}(J, \pi)}{\text{maxMK}(J, \pi)} \cdot 100$$

and in a similar way we also determined the errors $E_{\text{unif}}^2(J, \pi)$ and $E_{\text{unif}}^3(J, \pi)$ as follows:

$$E_{\text{unif}}^2(J, \pi) \overset{\text{def}}{=} \frac{\overline{ms}_{\text{unif}}^2(J, \pi) - \text{maxMK}(J, \pi)}{\text{maxMK}(J, \pi)} \cdot 100$$

$$E_{\text{unif}}^3(J, \pi) \overset{\text{def}}{=} \frac{\overline{ms}_{\text{unif}}^3(J, \pi) - \text{maxMK}(J, \pi)}{\text{maxMK}(J, \pi)} \cdot 100$$

The errors $E_{\text{unif}}^1(J, \pi)$, $E_{\text{unif}}^2(J, \pi)$ and $E_{\text{unif}}^3(J, \pi)$ are displayed in Figure 2.25a relative to the corresponding $\lambda_{\pi}$. The horizontal black line is the error “E_EXACT_MAKESPAN” of maxMK$(J, \pi)$ over the exact value of the maximum makespan. Obviously, this error is always 0. Also, for every speed assignment of $\pi$, we define the estimator $\overline{ms}_{\text{unif}}^{\text{min}}(J, \pi)$ as

$$\overline{ms}_{\text{unif}}^{\text{min}}(J, \pi) \overset{\text{def}}{=} \min \{ \overline{ms}_{\text{unif}}^1(J, \pi), \overline{ms}_{\text{unif}}^2(J, \pi), \overline{ms}_{\text{unif}}^3(J, \pi) \}$$

and its associated error

$$E_{\text{unif}}^{\text{min}}(J, \pi) \overset{\text{def}}{=} \frac{\overline{ms}_{\text{unif}}^{\text{min}}(J, \pi) - \text{maxMK}(J, \pi)}{\text{maxMK}(J, \pi)} \cdot 100$$

This error is displayed in Figure 2.25b relative to the corresponding $\lambda_{\pi}$. Finally, table 2.4 provides the reader with some statistics issued from the simulation.
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

(a) The three estimation errors $E_{\text{unif}}^1(J, \pi)$, $E_{\text{unif}}^2(J, \pi)$ and $E_{\text{unif}}^3(J, \pi)$ displayed relative to the corresponding $\lambda_{\pi}$.

(b) The estimation error $E_{\text{min\_unif}}^\text{exact}(J, \pi)$ displayed relative to the corresponding $\lambda_{\pi}$.

Figure 2.25: Simulation results.
2.13 Validity tests at a glance

<table>
<thead>
<tr>
<th></th>
<th>$E_{\text{unif}}^1(J, \pi)$</th>
<th>$E_{\text{unif}}^2(J, \pi)$</th>
<th>$E_{\text{unif}}^3(J, \pi)$</th>
<th>$E_{\text{unif}}^\text{min}(J, \pi)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>1.57%</td>
<td>1.89%</td>
<td>2.7%</td>
<td>1.57%</td>
</tr>
<tr>
<td>1st Qu.</td>
<td>6%</td>
<td>21.74%</td>
<td>13.28%</td>
<td>5.3%</td>
</tr>
<tr>
<td>Median</td>
<td>12.72%</td>
<td>41.07%</td>
<td>27.11%</td>
<td>9.92%</td>
</tr>
<tr>
<td>Mean</td>
<td>13.68%</td>
<td>37.91%</td>
<td>29.25%</td>
<td>10.44%</td>
</tr>
<tr>
<td>3rd Qu.</td>
<td>20.72%</td>
<td>55.5%</td>
<td>43.99%</td>
<td>15.08%</td>
</tr>
<tr>
<td>Maximum</td>
<td>32.96%</td>
<td>88.78%</td>
<td>68.01%</td>
<td>22.89%</td>
</tr>
<tr>
<td>Variance</td>
<td>69.76</td>
<td>359.37</td>
<td>320.47</td>
<td>33.36</td>
</tr>
<tr>
<td>Squared distance</td>
<td>8.35%</td>
<td>18.96%</td>
<td>17.9%</td>
<td>5.78%</td>
</tr>
<tr>
<td>Bias</td>
<td>42.35%</td>
<td>133.69%</td>
<td>94.05%</td>
<td>26.93%</td>
</tr>
<tr>
<td>Mean squared error</td>
<td>1863.25</td>
<td>18233.15</td>
<td>9165.97</td>
<td>758.43</td>
</tr>
</tbody>
</table>

Table 2.4: Statistics issued from the simulation

For obvious reason, the most accurate estimator (i.e., the most accurate upper-bound on the maximum makespan) is $\overline{m}_{\text{unif}}^\text{min}(J, \pi)$. As presented in Table 2.4, the most important error that we obtained for $\overline{m}_{\text{unif}}^\text{min}(J, \pi)$ is 22.89% and the minimal one is 1.57%. The average error is 10.44% with a squared distance of 5.78%. Hence, we believe that this is a promising path to go for more competitive bounds and for practical use. An open question remains however. For $\lambda_{\pi} \in [0, 2]$, we can see thanks to Figure 2.25a that $\overline{m}_{\text{unif}}^\text{min}(J, \pi) = \overline{m}_{\text{unif}}^1(J, \pi)$. Within this interval [0, 2], when the parameter $\lambda_{\pi}$ reaches an integer value (here, 1 and 2), something happens that considerably improves the accuracy of $\overline{m}_{\text{unif}}^\text{min}(J, \pi)$. But up to now, we do not have found any interpretation to that phenomenon.

2.13 Validity tests at a glance

This section provides a summary of all our validity tests. Through the chapter, we introduced many notations and this led us to sometimes present some tests in a particular context, i.e., the expression of these tests was written while considering a given set $J$ of jobs as input, rather than a multimode application. This is the reason why we present here the “final” expressions of these tests, in the sense that they are written in such a manner that their input is a multimode application $\tau$ composed of several set of
sporadic and constrained-deadline tasks. We thus present here one final test for each context listed below:

- Identical platforms and weakly work-conserving FJP schedulers
- Identical platforms and weakly work-conserving FTP schedulers
- Uniform platforms and strongly work-conserving FJP schedulers
- Uniform platforms and strongly work-conserving FTP schedulers

**Identical platforms, weakly work-conserving FJP schedulers:** For any multimode real-time application $\tau$ and any identical platform $\pi$ composed of $m$ processors, the protocol SM-MSO is valid provided that, for every mode $M^i$,

$$
\min\left\{ \overline{ms}^{\text{ident}}(J_i^{\text{wc}}, \pi), \overline{ms}_3^{\text{unif}}(J_i^{\text{wc}}, \pi) \right\} \leq \min_{j \neq i} \left\{ \min_{k=1}^{n_j} \{ D^j_k(M^i) \} \right\}
$$

where

1. $J_i^{wc}$ is the set of jobs that contains one job $J_k$ for each task $\tau_k^i$ and the processing time $c_k$ of each such job $J_k$ is $C_k^i$. That is, $J_i^{wc}$ contains $n_i \overset{\text{def}}{=} n_i$ jobs (where $n_i$ is the number of tasks of the mode $M^i$) and we assume that $J_i^{wc}$ is ordered by non-decreasing job processing times, i.e., $c_1 \leq c_2 \leq \cdots \leq c_n$.

2. $\overline{ms}^{\text{ident}}(J_i^{\text{wc}}, \pi)$ is defined as

$$
\overline{ms}^{\text{ident}}(J_i^{\text{wc}}, \pi) \overset{\text{def}}{=} \begin{cases} 
c_n & \text{if } n = m \\
\frac{\sum_{j=1}^{n-1} c_j}{m} + c_n & \text{otherwise}
\end{cases}
$$

3. $\overline{ms}_3^{\text{unif}}(J_i^{\text{wc}}, \pi)$ is defined as

$$
\overline{ms}_3^{\text{unif}}(J_i^{\text{wc}}, \pi) \overset{\text{def}}{=} \sum_{\ell=1}^{n} \left( c_\ell + \frac{1}{m^2} \cdot \sum_{j=1}^{\ell-1} c_j \right) \cdot \left(1 - \frac{1}{m}\right)^{n-\ell}
$$
2.13 Validity tests at a glance

**Identical platforms, weakly work-conserving FTP schedulers:** For any multimode real-time application \( \tau \) and any identical platform \( \pi \) composed of \( m \) processors, the protocol SM-MSO is valid provided that, for every mode \( M' \),

\[
\max_{k=1}^{m} \{ \text{Work}^{n_i+1}_k (J^\text{wc}_i) \} \leq \min_{j \neq i} \left\{ \min_{k=1}^{n_j} \{ D'_k(M') \} \right\}
\]

where

1. \( J^\text{wc}_i \) is the set of jobs that contains one job \( J_k \) for each task \( \tau^i_k \) and the processing time \( c_k \) of each such job \( J_k \) is \( C^i_k \). That is, \( J^\text{wc}_i \) contains \( n \overset{\text{def}}{=} n_i \) jobs (where \( n_i \) is the number of tasks of the mode \( M^i \)) and we assume that \( J^\text{wc}_i \) is ordered by decreasing \( S^i \)-priority.
2. \( \text{Work}^{n_j}_k (J^\text{wc}_i) \) is defined \( \forall k \in [1,m] \) and \( \forall j \in [1,n_i] \) as

\[
\text{Work}^{n_j}_k = \begin{cases} 
\text{Work}^{j-1}_k + c_j & \text{if } k = \max \{ \text{argmin} \{ \text{Work}^{j-1}_\ell \} \} \\
\text{Work}^{j-1}_k & \text{otherwise}
\end{cases}
\]

and \( \text{Work}^{0}_k = 0 \) \( \forall k \in [1,m] \).

**Uniform platforms, strongly work-conserving FJP schedulers:** For any multimode real-time application \( \tau \) and any uniform platform \( \pi = [s_1, s_2, \ldots, s_m] \) such that \( s_k \geq s_{k-1} \) \( \forall k \in [2,m] \), the protocol SM-MSO is valid provided that, for every mode \( M^i \),

\[
\min \left\{ \text{ms}_1^{\text{unif}} (J^\text{wc}_i, \pi), \text{ms}_2^{\text{unif}} (J^\text{wc}_i, \pi), \text{ms}_3^{\text{unif}} (J^\text{wc}_i, \pi) \right\} \leq \min_{j \neq i} \left\{ \min_{k=1}^{n_j} \{ D'_k(M') \} \right\}
\]

where

1. \( J^\text{wc}_i \) is the set of jobs that contains one job \( J_k \) for each task \( \tau^i_k \) and the processing time \( c_k \) of each such job \( J_k \) is \( C^i_k \). That is, \( J^\text{wc}_i \) contains \( n \overset{\text{def}}{=} n_i \) jobs (where \( n_i \) is the number of tasks of the mode \( M^i \)) and we assume that \( J^\text{wc}_i \) is ordered by non-decreasing job processing times, i.e., \( c_1 \leq c_2 \leq \cdots \leq c_n \).
2. \( \text{ms}_1^{\text{unif}} (J^\text{wc}_i, \pi) \) is defined as

\[
\text{ms}_1^{\text{unif}} (J, \pi) \overset{\text{def}}{=} \frac{\sum_{j=1}^{n} c_j}{s_m} - \frac{\sum_{k=1}^{m-1} \left( \sum_{j=1}^{n-m+k} c_j \cdot s_k \right)}{s_m \cdot s(1)}
\]
3. $ms^\text{unif}_2(J, \pi)$ is defined as

$$ms^\text{unif}_2(J, \pi) \equiv \frac{1}{s_m} \cdot \sum_{k=1}^{n} \left( c_k + s_1 \cdot \frac{\sum_{j=1}^{k-1} c_j}{s(1)} \right) \cdot K_{n-k}$$

where $K_j$ is such that $\forall j,$

$$K_j \equiv \begin{cases} 
1 & \text{if } s_1 = s_m \text{ and } j = 0 \\
\left(1 - \frac{s_1}{s_m}\right)^j & \text{otherwise}
\end{cases}$$

4. $ms^\text{unif}_3(J, \pi)$ is defined as

$$ms^\text{unif}_3(J, \pi) \equiv \frac{1}{s_m} \cdot \sum_{\ell=1}^{n} \left( c_\ell + \frac{s_\ell \cdot s_m}{s(1) \cdot \sum_{j=1}^{\ell-1} s_j} \cdot \sum_{j=1}^{\ell-1} c_j \right) \cdot H_{n-\ell}$$

where $x$ is such that $\frac{s_x}{\sum_{j=1}^{\ell} s_j} = \min_{k=1}^{n} \left\{ \frac{s_k}{\sum_{j=1}^{\ell} s_j} \right\}$ and $H_j$ is such that $\forall j,$

$$H_j \equiv \begin{cases} 
1 & \text{if } s_x = \sum_{i=1}^{\ell} s_i \text{ and } j = 0 \\
\left(1 - \frac{s_x}{\sum_{j=1}^{\ell} s_j}\right)^j & \text{otherwise}
\end{cases}$$

Uniform platforms, strongly work-conserving FTP schedulers: For any multimode real-time application $\tau$ and any uniform platform $\pi = [s_1, s_2, \ldots, s_m]$ such that $s_k \geq s_{k-1}$ $\forall k \in [2, m],$ the protocol SM-MSO is valid provided that, for every mode $M_i,$

$$\text{idle}^n_m(J_i^{\text{wc}}, \pi, S^i) \leq \min_{j \neq i} \left\{ \frac{n_j}{\min_{k=1}^{n_j} D_k^j(M_i)} \right\}$$

where

1. $J_i^{\text{wc}}$ is the set of jobs that contains one job $J_k$ for each task $\tau_k^i$ and the processing time $c_k$ of each such job $J_k$ is $C_k.$ That is, $J_i^{\text{wc}}$ contains $n_i \equiv n_i$ jobs (where $n_i$ is the number of tasks of the mode $M_i$) and we assume that $J_i^{\text{wc}}$ is ordered by decreasing $S^i$-priority.
2.14 Conclusion and future work

In this chapter, we addressed the scheduling problem of multimode real-time applications upon uniform multiprocessor platforms. We assumed that every mode of the application was scheduled by a global and fixed job-level priority scheduling algorithm. Under these assumptions, we proposed two protocols for managing every mode transition during the execution of multimode real-time applications on multiprocessor platforms. The first proposed protocol SM-MSO is synchronous in the sense that the tasks of the old- and new-mode are not scheduled simultaneously. The second protocol AM-MSO is asynchronous since it allows old- and new-mode tasks to be scheduled together. For both protocols, we established validity tests that allow the system designer to predict whether the given application can meet all the expected timing requirements upon the given platform. We prove the correctness of our schedulability analyses by extending the theory about the makespan determination problem. Our study focuses first on the particular case of identical multiprocessors platforms. Then, we address the more complex issue of uniform platforms for which we also provide schedulability analyses for both protocols SM-MSO and AM-MSO. For both identical and uniform platform models, this chapter addressed the scheduling problem for both Fixed-Task-Priority and Fixed-Job-Priority schedulers.

In our future work, we aim to take into account mode-independent tasks, i.e., tasks whose the periodic (or sporadic) activation pattern is not affected by the mode changes. Moreover, instead of scheduling the rem-jobs by using the scheduler of the old-mode during the transitions, it could be better, in term of the enablement delays applied to the new-mode tasks, to propose a dedicated priority assignment which meets the deadline of every rem-job, while minimizing the makespan. To the best of our knowledge, the problem of minimizing the makespan while meeting job deadlines together is not yet addressed in the literature and remains for future work.

 idle^\text{out}\text{m}_i(\mathcal{J}_i^{\text{wc}}, \pi, S') \text{ is computed as } idle^\text{out}_m \text{ where}

 idle^i_j = \begin{cases} 
 idle^{i-1}_j & \text{if } idle^{i-1}_j = idle^{i-1}_{j+1} \\
 idle^{i-1}_{j+1} & \text{else if } c_i \geq \sum_{k=1}^{j} (idle^{i-1}_{k+1} - idle^{i-1}_k) \cdot s_k \\
 idle^{i-1}_j + \frac{(c_i - \sum_{k=1}^{j} (idle^{i-1}_{k+1} - idle^{i-1}_k) \cdot s_k)}{s_j} & \text{otherwise}
\end{cases}

2.14 Conclusion and future work
CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS

References


REFERENCES


CHAPTER 2. SCHEDULING MULTIMODE REAL-TIME APPLICATIONS


Reducing the energy consumption by optimizing the hardware design

« Il est hélas devenu évident aujourd’hui que notre technologie a dépassé notre humanité. »

Albert Einstein

Contents

3.1 Context of the problem .............................................. 201
3.2 Model of computation .............................................. 204
3.3 Offline task mapping algorithms ................................. 211
3.4 Simulation results ................................................. 226
3.5 How to handle mode changes? ................................. 230
3.6 Conclusion and future work ..................................... 235
CHAPTER 3. OPTIMIZING THE HARDWARE DESIGN

Abstract

Nowadays, most of the energy-aware real-time schedulers use the Dynamic Voltage and Frequency Scaling (DVFS) feature of the processors to reduce the energy consumption of the system. These DVFS algorithms are usually efficient but, in addition to often consider unrealistic assumptions, they do not take into consideration the current evolution of the processor energy consumption profiles. In this chapter, we introduce a complement to the DVFS framework which preserves energy while considering the emerging technologies. This complementary approach focuses on the leakage consumption of the processors, on the contrary to classical DVFS techniques that focus only on the dynamic consumption. This chapter proposes a particular hardware design, together with appropriate software solutions. This hardware design is a multiprocessor platform composed of two kinds of processors and is compatible with any processor architecture, on the contrary to DVFS approaches that assume particular processor features. Then, the software methods that we propose make an efficient use of our hardware design and provide significant energy savings while reducing the complex energy-aware real-time scheduling problem to a simple and well-known bin-packing problem.
3.1 Context of the problem

3.1.1 Motivation

As introduced in Section 1.5.3 of Chapter 1, among all the hardware and software investigations launched into the concern of low-power system design, the “Dynamic Voltage and Frequency Scaling” (DVFS) [12] framework has become a major issue for energy-aware computer systems. This framework consists in minimizing the system energy consumption by managing both the supply voltage and the operating clock frequency of the processors. Targeting hard real-time systems, it focuses on doing so while respecting all the timing constraints. That is, DVFS approaches select the appropriate frequency and voltage of the processor(s) either at run- or design-time in order to reduce the consumption as much as possible while meeting all the system timing requirements. Thanks to the DVFS framework, the energy consumption of computer systems can be significantly reduced. This framework will be thoroughly discussed in Chapter 4.

Unfortunately, most studies about the DVFS framework suffer from a few drawbacks. First, the authors often consider unrealistic assumptions. For instance, it is often assumed that decreasing the operating frequency of a processor always reduces its energy consumption. But this not necessarily true in practice since several operating frequencies may be associated to the same supply voltage, and we saw in Section 1.4.4.5 of Chapter 1 that reducing the frequency without reducing the supply voltage is useless regarding the energy consumption. Also, It is often assumed that the frequency of a processor may be set to any non-integer values, or the time delay involved by a modification of the supply voltage is negligible. Although these assumptions are not too restrictive (since there exist some additional mechanisms that can bypass them to a certain extent), they can lead, however, to serious incompatibilities with the practical world if they are excessively exploited. Then, DVFS strategies obviously require DVFS-capable processors. Although many such processors have been proposed over the years, only a few of them have been used in embedded systems. One major reason is that many DVFS processors involve large mass production cost including test cost, design cost and the cost of on-chip DC-DC converters [36]. These drawbacks are just some examples of hindrances to the deployment of DVFS techniques into the practical world.

The main drawback of this framework is that most of the DVFS techniques do not
CHAPTER 3. OPTIMIZING THE HARDWARE DESIGN

take into account the leakage power dissipation $P_{\text{leak}}$ of the processors, due to leakage and subthreshold currents (see Section 1.4.4.3 on page 48 for more details about this undesirable power dissipation). That is, DVFS techniques only reduce the dynamic consumption of the processors (due to the power dissipation $P_{\text{dyn}}$ and $P_{\text{short}}$ described on page 48). As explained in this previous chapter, the leakage consumption has become more and more significant as feature sizes have become smaller and threshold levels lower [17]. Therefore, since the dynamic consumption of the processors becomes less important as the leakage consumption grows, the energy savings provided by the DVFS algorithms has become less and less significant over the years. Nowadays, as technologies scale down to the UDSM (Ultra Deep-SubMicron), the leakage consumption becomes more dominant than the dynamic consumption [44], and can no longer be neglected. Note that, up to now, the DVFS framework has provided significant energy savings in practice; this is the reason why this framework has been so widely studied in the literature. Hence, we insist on the fact that our intention in this chapter is not to deny the efficiency of the DVFS framework (this would be strongly unbecoming since we also propose some DVFS algorithms in Chapter 4). Rather, we want to emphasize the fact that, in future studies which will target modern technologies, the leakage consumption should be taken into consideration at least as much as the dynamic consumption.

Although this chapter does not introduce new scheduling algorithms or novel formal proofs, it aims to address new solutions to the emerging problem of the leakage consumption. As it was already mentioned in 2006 [17]: « leakage power dissipation is becoming a huge factor challenging a continuous success of CMOS technology in the semiconductor industry (…) innovations in leakage control and management are urgently needed. ». We propose in this chapter a complement to the DVFS framework that focuses on the leakage consumption. The three objectives of our proposal are:

1. be compatible with any processor architecture (not only with DVFS processors as the DVFS framework),
2. provide important energy savings by making an efficient use of the features of our proposed platform model and the characteristics of the real-time application,
3. considerably simplify the complex energy-aware scheduling problem, in regard with the existing solutions cited in the next section.

On the contrary to the existing solutions that are typically computational-intensive or/and provide low energy savings, our methodology is based on some well-known optimization heuristics and may therefore benefit (depending on the selected parameters
of the heuristics) from a low complexity while providing significant energy savings.

3.1.2 Related works

In the scheduling literature, the energy-aware scheduling problem upon uniform platforms is generally divided into two sub-problems [23]: the Task Scheduling and Voltage Scaling problem (TSVS) and the The Task Mapping Improvement problem (TMI).

The TSVS problem. The TSVS problem assumes that a mapping of the tasks upon the processors is known beforehand, where each task is assigned to only one processor. For a given task mapping, this problem consists in determining the schedule/order of the tasks executions on each CPU and their assignment to the various voltage levels so as to minimize the total energy consumption. Currently, most of the papers (see [25, 26, 27, 28, 34, 35, 38] for instance) focus on this TSVS problem, while considering different task models. In [46], the authors formulate the task mapping and TSVS problems as a mixed integer linear programming with discrete CPU voltage levels and propose a relaxation heuristic. There are also few papers (see [2, 41, 47] for instance) that assume that the task mapping and task ordering are known beforehand and focus only on the voltage scaling problem.

The TMI problem. The TMI problem consists in iteratively improving the task mapping, based on the system schedulability and the energy consumption of the produced schedule.

The authors of [23] consider the TSVS and TMI problems together in order to derive a feasible and energy-efficient schedule. Leung et al. [32] formulate the whole problem of task mapping, task ordering and voltage scaling as a mixed integer non-linear programming problem with continuous CPU voltage levels. In [40] Schmitz et al. propose a strategy that also considers task mapping, task ordering and voltage scaling, where the priorities of the tasks are generated using a genetic algorithm and voltage scaling of the tasks is done using the method proposed in [41].

3.1.3 Chapter organization

This chapter is structured as follows. Section 3.2 defines the computational models used throughout the chapter. In particular, this section introduces our platform model. Section 3.3 introduces our non-DVFS methodology and explains how our proposed
platform and methodology reach the three objectives formulated on page 201. Then, we formalize the considered problem and we address the multiple issues of optimal and approximated approaches to solve it. Section 3.4 reports on some simulation results conducted using the methods introduced in Section 3.3 and finally, Section 3.6 gives our conclusions and introduces our future work.

3.2 Model of computation

3.2.1 Hardware model

In this section, we introduce our dual CPU type Multiprocessor System-on-Chip (MPSoC) platform illustrated in Figure 3.1 and referred to as DMP in the following. MPSoC systems are widely used in the design of high-performance and low-power embedded systems as it has been already suggested in the literature [30]. A DMP is composed of two MPSoC platforms denoted $L_{lp}$ and $L_{hp}$ containing $m_{lp}$ and $m_{hp}$ processors respectively. Each processor has a small amount of local memory (L1 cache memory), allowing the processor for operating independently, at least for a certain period of time, before fetching the data from the possible L2 cache memory and/or distant (off-chip) central memory. All the processors in the DMP are identical in terms of their architecture, i.e. their hardware specifications are derived from the same Register Transfer Logic (RTL) description in some Hardware Description Language (HDL), such as VHDL or Verilog. However, the processors of each platform differ from those of the other platform in the standard-cell library used during their physical implementation (see Chapter 1 on page 40 for details). The processors in $L_{lp}$ (called the lp-processors hereafter) are manufactured using a Low-Power, Low-Performance standard-cell library. The resulting integrated circuit can be operated with lower power supply voltages and consequently lower operating clock frequency, resulting in lower dynamic power dissipation $Pwr_{dyn}$. Inversely, the processors in $L_{hp}$ (called the hp-processors from this point forward) are physically implemented using a High-Power, High-Performance standard-cell library (for the same or different technological node). Such circuit allows higher operating frequency, but will require higher supply voltage, resulting in higher dynamic power dissipation $Pwr_{dyn}$ than the lp-processors. In this way, while each MPSoC platform can be seen as identical multiprocessor platform, the DMP as a whole appears as a particular case of a uniform multiprocessor platform. Hardware architectures gathering different processor technologies are sometimes referred to as “heterogeneous” multiprocessor platforms in the literature (see [42, 43] for instance) and such designs offer the best
solution and trade-offs with respect to software programmability, flexibility, silicon area and power consumption [43]. Notice that in our model, there is no restriction on the processor purposes, hence achieving our first objective introduced on page 202.

![Diagram of DMP model](image)

**Figure 3.1:** Our considered DMP model with $2 \times (5 \times 5)$ processors.

Different processors, within the same MPSoC platform, communicate between themselves using the Network-on-Chip (NoC) communication paradigm. In recent years, the NoCs have gained many interests in the research community and are more and more used to replace traditional communication solutions based on shared bus and their variates [9, 14]. When compared to buses, the NoCs profit from higher bandwidth capacities because of their concurrent operation and much lower and predictable latencies, allowing efficient processor-to-processor communication in our case. NoCs are flexible, scalable and even energy efficient [11, 45] for systems containing more than ten nodes (processors and/or memories).

The communication between different nodes using NoCs is quite simple. The raw data, coming from a master node (node capable to issue read or write operation) is first
CHAPTER 3. OPTIMIZING THE HARDWARE DESIGN

transformed into NoC protocol packets, in the master Network Interface unit (node to NoC protocol conversion). These packets will travel through a certain number of routers before reaching their final destination, a slave Network Interface side, where the raw data is extracted from packets and delivered to the slave node.

In DMP, each MPSoC is built using regular, fully connected mesh NoC. That is, each microprocessor is connected to one router through a Network Interface. Each router is connected to its 4 direct neighbors (note that the routers at the edges are connected, so that they form a torus). Of course other NoC topologies could be explored, for more efficient (lower latency) inter processor communication, but this is out of the scope of this research. The two MPSoC platforms are assumed to be independent and they cannot communicate directly. However, since they share the same central memory, the communication is still possible, although not in a very efficient way. But this is not critical in this “first-step” research since we assume in the following that the two MPSoC platforms do not communicate. A thorough reflection about the many possible communication technologies between the two MPSoC $L_{lp}$ and $L_{hp}$ should be addressed and exploited in future works.

An $x$-processor (where $x$ is either $lp$ or $hp$) is characterized by two parameters: its estimated power dissipation $P_{\text{run}}^x$ while it is running a task at its maximal clock frequency; and its estimated power dissipation $P_{\text{idle}}^x$ while it idles. Notice that in this work, all the supplied processors run at their maximal operating clock frequency and both their supply voltage and clock frequency are never modified.

3.2.2 Application model

All the algorithms proposed in this chapter are compatible with the multimode real-time application model introduced in Chapter 1, on page 20. But they are designed in such a manner that they are performed successively on each operating mode separately. That is, they provide energy savings to each mode separately and are not concerned about the mode transitions, i.e., no energy saving is provided during mode transitions. The problem of transitioning from one mode to another one while using the approaches presented here will be thoroughly discussed at the end of this chapter (see Section 3.5, page 230).
3.2 Model of computation

Since the methods presented in this chapter focus only on individual operating modes, this work considers (for sake of both clarity and notations readability) the scheduling of a single set of \( n \) sporadic constrained-deadline tasks, according to the interpretations given in Chapter 1, Section 1.2.3. However, we extend here the notations in order to cope with our particular DMP platform model. Each task \( \tau_i = (C^{lp}_i, C^{hp}_i, D_i, T_i) \) is characterized by four parameters—a worst-case execution time at maximal frequency \( C^{lp}_i \) and \( C^{hp}_i \) upon the lp- and hp-processors (respectively), a minimal inter-arrival delay \( T_i \) and a relative deadline \( D_i \leq T_i \)—with the interpretation that the task generates successive jobs \( \tau_{i,j} \) (with \( j = 1, \ldots, \infty \)) released at times \( a_{i,j} \) such that \( a_{i,j} \geq a_{i,j-1} + T_i \) (with \( a_{i,1} \geq 0 \)), each such job has an execution requirement of at most \( C^{lp}_i \) or \( C^{hp}_i \) (depending on the MPSoC upon which it executes, i.e., \( L_{lp} \) or \( L_{hp} \)) and must be completed by its absolute deadline \( D_{i,j} = a_{i,j} + D_i \). Finally, the tasks are assumed to be independent in the sense that they do not share any resource (except the processors), they do not communicate and there are no precedence constraints between them. Below, we introduce some notions necessary for the remainder of this chapter and we adapt the notions of task utilization and task density (previously defined on page 15) to our hardware model.

**Definition 3.1 (Task utilization)**

We define the utilization \( U^x_i \) of a task \( \tau_i \) upon an \( x \)-processor as the ratio between its worst-case execution time upon an \( x \)-processor running at maximal frequency and its period, i.e.,

\[
U^x_i \overset{\text{def}}{=} \frac{C^x_i}{T_i}
\]

**Definition 3.2 (Generalized utilization and maximal utilization)**

The generalized utilization \( U^x_{\text{sum}}(\tau) \) and the maximal utilization \( U^x_{\text{max}}(\tau) \) of a set of tasks \( \tau \) allocated to a MPSoC platform \( L_x \) are defined as follows:

\[
U^x_{\text{sum}}(\tau) \overset{\text{def}}{=} \sum_{\tau_i \in \tau} U^x_i
\]

and

\[
U^x_{\text{max}}(\tau) \overset{\text{def}}{=} \max_{\tau_i \in \tau}(U^x_i)
\]
CHAPTER 3. OPTIMIZING THE HARDWARE DESIGN

**Definition 3.3 (Task density)**

We define the density $\delta^x_i$ of a task $\tau_i$ upon an $x$-processor as the ratio between its worst-case execution time upon an $x$-processor running at maximal frequency and its deadline, i.e.,

$$\delta^x_i \overset{\text{def}}{=} \frac{C^x_i}{D_i}$$

**Definition 3.4 (Generalized density and maximal density)**

The generalized density $\delta^x_{\text{sum}}(\tau)$ and the maximal density $\delta^x_{\text{max}}(\tau)$ of a set of tasks $\tau$ allocated to a MPSoC platform $L_x$ are defined as follows:

$$\delta^x_{\text{sum}}(\tau) \overset{\text{def}}{=} \sum_{\tau_i \in \tau} \delta^x_i$$

and $\delta^x_{\text{max}}(\tau) \overset{\text{def}}{=} \max_{\tau_i \in \tau}(\delta^x_i)$

We assume that $C^l_{i,\tau} \geq C^l_{j,\tau}$ if and only if $C^h_{i,\tau} \geq C^h_{j,\tau}$ $\forall \tau_i, \tau_j$ and without loss of generality, we assume that the tasks are indexed by decreasing order of their density, i.e. $\delta^x_1 \geq \delta^x_2 \geq \ldots \geq \delta^x_n$, and consequently $\delta^x_{\text{max}}(\tau) = \delta^x_1$. Notice that a task $\tau_i$ such that $\delta^l_i > 1$ cannot be completed in time while being executed upon a lp-processor. As a result, such tasks must mandatorily be assigned to the MPSoC platform $L_{hp}$ (assuming that $\delta^h_i \leq 1$ $\forall \tau_i$). The following definitions have been adapted from the definitions in [8].

**Definition 3.5 (Demand bound function)**

For any time interval of length $t$, the demand bound function $\text{DBF}^x(\tau_i, t)$ of a sporadic task $\tau_i$ allocated to a MPSoC platform $L_x$ provides an upper bound on the cumulative execution requirements by jobs of $\tau_i$ that are both released in, and have deadline within, any interval of length $t$. This demand bound function is given by

$$\text{DBF}^x(\tau_i, t) \overset{\text{def}}{=} \max \left\{ 0, \left[ \frac{t - D_i}{T_i} \right] + 1 \right\} \cdot C^x_i$$
3.2 Model of computation

**Definition 3.6 (Application load)**

The load parameter, based upon the \( \text{DBF} \) function, is defined for any set of tasks \( \tau \) allocated to a MPSoC platform \( L_x \) as follows:

\[
\text{LOAD}^x(\tau) \triangleq \max_{t > 0} \left\{ \sum_{\tau_i \in \tau} \frac{\text{DBF}^x(\tau_i, t)}{t} \right\}
\]

As mentioned in [6], the \( \text{DBF} \) and \( \text{LOAD} \) functions can be computed *exactly* (by the methods proposed in [8, 39]) or *approximately* (by those proposed in [3, 18, 19]) to any arbitrary degree of accuracy in pseudo-polynomial time or polynomial time\(^1\), respectively.

### 3.2.3 Scheduling specifications

Each MPSoC platform, \( L_{lp} \) or \( L_{hp} \), uses its own global multiprocessor scheduling algorithm. In this study, each task is statically assigned to one of the two MPSoC platforms (\( L_{lp} \) or \( L_{hp} \)) at system design-time. We assume that every job can start its execution on any processor of its assigned platform \( L_x \) and *may migrate at run-time* to any other processor of \( L_x \) (with no loss or penalty) if it gets meanwhile preempted by a higher-priority job. However, tasks and jobs are not allowed to migrate between the two platforms of the DMP since we assume that there is no communication between them.

We consider in this research two popular multiprocessor scheduling algorithms: the *preemptive global Deadline Monotonic* and the *preemptive global Earliest Deadline First* [4]. For sake of simplicity, these algorithms will henceforth be referred as global-DM and global-EDF. Recall that

- Global-DM is a FTP scheduler that assigns a static (i.e. constant) priority to each *task* at system design-time according to their relative deadlines: the smaller the deadline, the greater the priority. Ties are arbitrarily broken.

- Global-EDF is a FJP scheduler that assigns priority to *jobs* at run-time according

---

\(^1\)Notice that the computing complexity of the polynomial-time algorithm called “PTAS-\( \delta_{\text{sum}}(\tau, \epsilon) \)” in [18] can be improved. Indeed, it is not necessary to consider every time-instant \( t \in S(\tau, \epsilon) \) by increasing order, since the algorithm returns \( \max_{t \in S(\tau, \epsilon)} f^r(\tau, t) \).

209
CHAPTER 3. OPTIMIZING THE HARDWARE DESIGN

to their absolute deadlines: the earlier the deadline, the greater the priority. Again, ties are arbitrarily broken.

As mentioned in Section 1.3.5.1 (Chapter 1, page 29), there exist many other global schedulers and among the most popular ones, one can cite the “PFAIR-like” (Proportionate fair) strategies such as PFAIR [5], PD [7], PD\(^2\) [1]. The worst drawback of all Pfair-like approaches resides in the fact that tasks are rescheduled in every quantum of time (typically set to an arbitrary low value). That is, although the theoretical correctness of these algorithms PF, PD and PD\(^2\) is undeniable, their application in the real world is limited due to the high number of overheads (and especially the preemption overheads) that are generated during the execution of the system [15]. More recently, Zhu, et al., investigated the problem of the overheads and designed a new class of algorithms named “Boundary fair schedulers” (i.e., “Bfair-like” algorithms). Their idea is to ensure the fairness only at boundaries, i.e., some particular events occurring at run-time such as job releases, job deadline, etc., resulting in a considerable reduction of the number of decisions taken by the scheduler, as well as the cost of the overheads compared to Pfair-like algorithms. Among the most interesting studies in this area, one can cite [48], [13] and [20]. Basically, these Bfair-like schedulers can be divided into two categories: the “continuous-time” Bfair schedulers, i.e., the time is assumed to be continuous, and the “quantum-based” Bfair schedulers that divide the time into time slots. To the best of our knowledge, the two most recent approaches belonging to the continuous-time category are LRE-TL [20] and DP-WRAP [33]. These algorithms are optimal\(^1\) (as every scheduler in this category) but suffer from another drawback: they can sometimes preempt a task \(\tau_i\) by another task \(\tau_j\) only to execute an extremely small portion of \(\tau_j\). That is, the generated schedule can undergo a preemption overhead only to execute an insignificant portion of a task, hence producing schedules in which the time actually used to execute the tasks is limited. On the other hand, quantum-based techniques have been designed in order to ensure that the Operating System only preempts a task \(\tau_i\) by another task \(\tau_j\) if \(\tau_j\) has to execute at least one time-quantum. In our opinion, the most interesting study in this category is [48]. The proposed scheduler BF is optimal and it considerably reduces the number of preemptions compared to continuous-time Bfair-like and Pfair-like schedulers. However, the complexity of BF is pseudo polynomial; more precisely, the complexity is \(O(n \cdot T_{\text{max}})\) where \(T_{\text{max}}\) is the largest period of the tasks, but since \(T_{\text{max}}\) could be quite large, the overhead due to the execution of the scheduler can be large as well. Currently, we are working on an adaptation of BF that we named BF\(^2\) in [21]

\(^1\)An algorithm is said to be optimal if it produces a feasible schedule as long as such a feasible schedule exists.
by analogy with PD and PD². Our goal while designing BF² is to actually reduce the
time-complexity of BF to O(n) and to take into account into the schedulability analysis
the various types of overhead due to hardware and software interruptions. Although
all these techniques cited above are able to schedule tasks to meet deadlines even when
up to 100% of the processing capacity is requested, they support only implicit-deadline
tasks. Since sporadic tasks are considered in this chapter, we limited this study to FTP
and FJP schedulers.

### 3.3 Offline task mapping algorithms

#### 3.3.1 Our methodology

This research focuses on the energy consumption of a DMP while executing real-time
applications. Since a task consumes a lower amount of energy while executed upon
L_{lp} than upon L_{hp}, the main idea for minimizing the energy consumption of a DMP is
to maximize the workload onto L_{lp}. We thus address the problem of how to partition,
at system design-time, the set of real-time tasks upon the two MPSoC platforms L_{lp} and L_{hp},
so that the tasks are completed in time with an energy consumption as low as possible. The
mapping of the tasks that we determine assigns each task to only one MPSoC platform
(either L_{lp} or L_{hp}), not to a specific processor. That is, any task mapping splits the set of
tasks \( \tau \) into two subsets \( \tau^{lp} \) and \( \tau^{hp} \), each one designed to be scheduled on platform L_{lp}
and L_{hp}, respectively. Then, once the task mapping is established, only a subset of CPUs
are powered on in both L_{lp} and L_{hp}. The amount of CPUs powered on in each platform
L_x (x is either lp or hp) must ensure that the set of tasks \( \tau^x \) assigned to that platform
can be successfully scheduled (i.e., without missing any deadline) during the execution
of the application. In both platforms, all the processors that are not strictly required to
fulfill the timing requirements are turned off in order to nullify their leakage power dissipation.
For a given task mapping, the resulting hardware configuration of the DMP is
therefore energy-optimized for the targeted real-time application (hence achieving the
second objective introduced in Section 3.1.1). At run-time, the tasks are scheduled upon
their assigned platform by using a global scheduling algorithm (possibly different on
each platform). Therefore, any DVFS algorithm targeting identical platforms (including
those proposed in the next chapter) may be used on both L_{lp} and L_{hp} to further reduce
their dynamic consumption.
CHAPTER 3. OPTIMIZING THE HARDWARE DESIGN

It can sound inadequate to definitively turn off some processors before the execution of the application. However, we can assume that for some kinds of practical applications, these unsupplied CPUs could be turned on later (at run-time) in order to replace a defective supplied processor, to handle an emergency, and so on. Moreover, this strategy of switching off some processors is also very useful for multimode real-time applications. For such applications, a task mapping can be determined for the set of tasks of each mode separately; that is, the task set of each mode is split into two subsets $\tau_{lp}$ and $\tau_{hp}$. Moreover, the approaches presented in this chapter associate to each task mapping a certain number of processors powered on (and off) in both MPSoC platforms $L_{lp}$ and $L_{hp}$. For each mode (and thus, for each corresponding task mapping $\{\tau_{lp}, \tau_{hp}\}$), the determined number of processors that are powered on in the platform $L_{lp}$ is sufficient to schedule the task set $\tau_{lp}$ without missing any deadline. Symmetrically, the determined number of processors that are powered on in the platform $L_{hp}$ is sufficient to successfully schedule the task set $\tau_{hp}$. For such multimode applications, the hardware configuration of the DMP is therefore energy-optimized for each mode of the application. Although the multimode application model is not explicitly used in this chapter, it justifies our hardware architecture and the fact that some CPUs are turned off. Figure 3.2 illustrates an example of a (fictive) multimode application where a task mapping is associated to each mode. In this figure, the squares depicted in green are the CPUs powered on while the CPUs powered off are represented by crossed-out squares.

When a mode change is requested at run-time, the system switches the task set of the current mode to the task set of the new-mode; that is, the task mapping $\{\tau_{lp}, \tau_{hp}\}$ of the current mode is replaced with the task mapping $\{\tau_{lp}, \tau_{hp}\}$ of the new-mode. However, since the task mapping of each mode has its own associated numbers of CPUs powered on (and off) in both platforms $L_{lp}$ and $L_{hp}$, deciding how many CPUs have to be powered on during the mode transitions is not straightforward and we will address this problem at the end of this chapter, on page 230. Initially (during the design of the system), we assume that the number of processors in each MPSoC platform is set to the maximum number of CPUs required by all the modes.

As mentioned earlier, this chapter does not focus on multimode applications (the discussion above was introduced only with the intention of justifying our methodology and our platform model). Rather, we assume here that the application is composed of a single set of tasks for which our approaches determine an appropriate partition.
3.3 Offline task mapping algorithms

Figure 3.2: Illustration of a multimode real-time application composed of 5 operating modes (the same as in Figure 1.7 on page 20). Here, a task mapping is determined beforehand for each mode. That is, the set of tasks of each mode is split into two subsets $\tau_{lp}$ and $\tau_{hp}$ and each subset is scheduled upon its respective platform $L_{lp}$ or $L_{hp}$ using only a sufficient number of CPUs. The squares depicted in green are the supplied CPUs while the CPUs turned off are represented by crossed-out squares. 

$\{\tau_{lp}, \tau_{hp}\}$. The schedulability of the two task sets $\tau_{lp}$ and $\tau_{hp}$ upon $L_{lp}$ and $L_{hp}$ (resp.) is asserted thanks to schedulability tests. Currently, several schedulability tests have already been established for the global scheduling algorithms, the task model and the identical MPSoC platform models considered in this chapter. From a practical point of view, the use of a global scheduling algorithm on each platform $L_{lp}$ and $L_{hp}$ implies that both of them may be seen as a single computing element. The task mapping determination problem may therefore be compared to a bin-packing problem with two bins (the two platforms $L_{lp}$ and $L_{hp}$) and $n$ items (the $n$ real-time tasks), hence achieving the third objective introduced in Section 3.1.1, page 201. In our simulations in Section 3.4, we will show that even some popular bin-packing heuristics may provide task mappings that generate significant energy savings.
CHAPTER 3. OPTIMIZING THE HARDWARE DESIGN

3.3.2 Notations

The task mapping determination is thus the process of determining, at system design-time, the partitioning of the task set into two subsets of tasks $\tau_{lp}$ and $\tau_{hp}$ that leads to a minimal (or so) consumption when $\tau_{lp}$ and $\tau_{hp}$ are scheduled upon $L_{lp}$ and $L_{hp}$, respectively. These two subsets of tasks are exhaustive and mutually exclusive, i.e., $\tau_{lp} \cup \tau_{hp} = \tau$ and $\tau_{lp} \cap \tau_{hp} = \phi$. The subset $\tau_{lp}$ contains the tasks executed upon the MPSoC platform $L_{lp}$ and symmetrically, $\tau_{hp}$ contains the tasks executed upon $L_{hp}$. We call such partition $\{\tau_{lp}, \tau_{hp}\}$ a task mapping.

We denote by $S_{lp}$ and $S_{hp}$ the global scheduling algorithms used on $L_{lp}$ and $L_{hp}$, respectively (where $S_{lp}$ and $S_{hp}$ are either global-DM or global-EDF), and we will use the notation $S^x$ to denote any of these two scheduling algorithms. We denote by $\tau$ any set of sporadic constrained-deadline tasks and by $\text{cru}_{x}^S(\tau)$ the function that returns a sufficient number of $x$-processors so that the task set $\tau$ can be scheduled by $S^x$ on $L_x$ without missing any deadline. Therefore, once a task mapping $\{\tau_{lp}, \tau_{hp}\}$ is determined for a given set $\tau$ of tasks, the numbers of processors powered on in $L_{lp}$ and $L_{hp}$ are given by $\text{cru}_{lp}^{S_{lp}}(\tau_{lp})$ and $\text{cru}_{hp}^{S_{hp}}(\tau_{hp})$, respectively. Unfortunately, no necessary and sufficient schedulability test is known for global-DM and global-EDF in order to determine the minimal number of required CPUs to schedule a given set of sporadic constrained-deadline tasks. Fortunately, sufficient tests exist, thus providing a sufficient number of supplied processors for each platform.

3.3.3 Schedulability test for global-DM

For the global-DM algorithm, the function $\text{cru}_{x}^{\text{DM}}(\tau)$ makes use of the following sufficient schedulability condition from [4].

**Schedulability Test 3.1 (From [4])**

A set of sporadic constrained-deadline tasks $\tau$ is guaranteed to be schedulable upon an identical multiprocessor platform $L_x$ composed of $m$ processors using preemptive global-DM if, for every task $\tau_k$,

$$\sum_{\tau_i \in \tau \text{ such that } D_i < D_k} \frac{Q_{iL}^x(\tau)}{1 - \delta_{k}^{x}} \leq m$$
3.3 Offline task mapping algorithms

where

\[
\alpha_{i,k}^{x}(\tau) = \begin{cases} 
U_i^x \cdot \left(1 + \frac{T_i - C_i}{D_k} \right) & \text{if } \delta_{\text{max}}^x(\tau) \geq U_i^x \\
U_i^x \cdot \left(1 + \frac{T_i - C_i}{D_k} \right) + \frac{C_i - \delta_{\text{max}}^x(\tau) \cdot T_i}{D_k} & \text{otherwise}
\end{cases}
\]

From Test 3.1, the function \( \text{crpu}^{DM}_x(\tau) \) can therefore be written as

\[
\text{crpu}^{DM}_x(\tau) \overset{\text{def}}{=} \max_{\tau_i \in \tau} \left\{ \sum_{\tau_i \in \tau \text{ such that } D_i < D_k} \alpha_{i,k}^{x}(\tau) \right\} \left(1 - \delta_k^x \right)
\]

where \( \alpha_{i,k}^{x}(\tau) \) is defined as in Test 3.1.

3.3.4 Schedulability test for global-EDF

The global-EDF algorithm has been widely studied in the literature and several (incomparable) sufficient schedulability conditions were already established. In this chapter, we use a combination of the most popular ones (Inequalities 3.1, 3.2 and 3.3) that we have gathered in Test 3.2. Notice that we have expressed these three inequalities in the number \( m \) of required processors.

\textbf{Schedulability Test 3.2 (From [4, 6, 10])}

A set of sporadic constrained-deadline tasks \( \tau \) is schedulable using global-EDF upon an identical multiprocessor platform \( L_x \) composed of \( m \) processors, provided BCL condition [10]:

\[
m \geq \frac{\delta_{\text{sum}}^x(\tau) - \delta_{\text{max}}^x(\tau)}{1 - \delta_{\text{max}}^x(\tau)} \tag{3.1}
\]

or BAK condition [4]:

\[
m \geq \max_{\tau_i \in \tau} \left\{ \sum_{\tau_i \in \tau} \min(1, \beta_{i,k}^x) - \delta_k^x \right\} \frac{1 - \delta_{\text{max}}^x(\tau)}{1 - \delta_k^x} \tag{3.2}
\]

where

\[
\beta_i^x = \begin{cases} 
U_i^x \cdot \left(1 + \frac{T_i - D_i}{D_k} \right) & \text{if } \delta_k^x \geq U_i^x \\
U_i^x \cdot \left(1 + \frac{T_i - D_i}{D_k} \right) + \frac{C_i - \delta_k^x \cdot T_i}{D_k} & \text{if } \delta_k^x < U_i^x
\end{cases}
\]
or BB condition [6]:

\[
m \geq \frac{\text{load}^x(\tau) - \delta_{\text{max}}^x(\tau)(1 - \delta_{\text{max}}^x(\tau))}{(1 - \delta_{\text{max}}^x(\tau))^2}
\]  

(3.3)

where \text{load}^x(\tau) is defined as in Section 3.2.2.

**Proof**

First, Expression 3.1 is derived from [10]. In this paper the authors prove that any set \( \tau \) of \( n \) sporadic constrained-deadline tasks is schedulable by global-EDF upon any identical multiprocessor platform \( L_x \) composed of \( m \) processor provided

\[
\delta_{\text{sum}}^x(\tau) \leq m \cdot (1 - \delta_{\text{max}}^x(\tau)) + \delta_{\text{max}}^x(\tau)
\]

Subtracting \( \delta_{\text{max}}^x(\tau) \) from both sides of the above inequality yields

\[
\delta_{\text{sum}}^x(\tau) - \delta_{\text{max}}^x(\tau) \leq m \cdot (1 - \delta_{\text{max}}^x(\tau))
\]

And finally, dividing both sides by \((1 - \delta_{\text{max}}^x(\tau))\) provides

\[
\frac{\delta_{\text{sum}}^x(\tau) - \delta_{\text{max}}^x(\tau)}{1 - \delta_{\text{max}}^x(\tau)} \leq m
\]

Note that we assumed on page 208 that every task \( \tau_i \) whose \( \delta_{lp}^i > 1 \) is assigned to \( L_{hp} \) and we also assumed that \( \delta_{hp} < 1 \ \forall \tau_i \). Consequently, the term \((1 - \delta_{\text{max}}^x(\tau))\) is non-negative.

Second, Expression 3.2 is derived from [4]. As for the previous expression, the authors showed that any set \( \tau \) of \( n \) sporadic constrained-deadline tasks is schedulable by global-EDF upon any identical multiprocessor platform \( L_x \) composed of \( m \) processor provided

\[
\forall \tau_k : \sum_{\tau_i \in \tau} \min\{1, \beta_{\tau_i}^x\} \leq m \cdot (1 - \delta_{\text{max}}^x(\tau)) + \delta_{\text{max}}^x(\tau)
\]

where \( \beta_{\tau_i}^x \) is defined as previously. Subtracting \( \delta_{\text{max}}^x(\tau) \) from both sides of the above inequality yields

\[
\forall \tau_k : \sum_{\tau_i \in \tau} \min\{1, \beta_{\tau_i}^x\} - \delta_{\tau_k}^x \leq m \cdot (1 - \delta_{\tau_k}^x)
\]

Then, by dividing both sides by \((1 - \delta_{\tau_k}^x)\) we get

\[
\forall \tau_k : \frac{\sum_{\tau_i \in \tau} \min\{1, \beta_{\tau_i}^x\} - \delta_{\tau_k}^x}{(1 - \delta_{\tau_k}^x)} \leq m
\]
3.3 Offline task mapping algorithms

And thus,

\[
\max_{\tau \in \mathcal{T}} \left\{ \frac{\sum_{i \in \tau} \min\{1, \beta_i^x\} - \delta_k^x}{1 - \delta_k^x} \right\} \leq m
\]

Finally, Expression 3.3 is derived from [6] where the authors proved that any set \( \tau \) of \( n \) sporadic constrained-deadline tasks is schedulable by global-EDF upon any identical multiprocessor platform \( L_x \) composed of \( m \) processors provided

\[
\text{load}^x(\tau) \leq \mu^x - (\lceil \mu^x \rceil - 1) \cdot \delta_{\max}^x(\tau)
\]

where \( \mu^x \overset{\text{def}}{=} m - (m - 1) \cdot \delta_{\max}^x(\tau) \). The above inequality can be rewritten as

\[
\text{load}^x(\tau) - \delta_{\max}^x(\tau) \leq \mu^x - \lceil \mu^x \rceil \cdot \delta_{\max}^x(\tau) \tag{3.4}
\]

Any set \( \tau \) of \( n \) sporadic constrained-deadline tasks is schedulable by global-EDF upon any identical \( m \)-processors multiprocessor platform \( L_x \) as long as the above inequality is verified. Therefore, since \( \lceil \mu^x \rceil < \mu^x + 1 \), the schedulability is still guaranteed as long as

\[
\text{load}^x(\tau) - \delta_{\max}^x(\tau) \leq \mu^x - (\mu^x + 1) \cdot \delta_{\max}^x(\tau)
\]

and thus it is needed that

\[
\frac{\text{load}^x(\tau)}{(1 - \delta_{\max}^x(\tau))} \leq \mu^x
\]

Since \( \mu^x = m - (m - 1) \cdot \delta_{\max}^x(\tau) \), the right-hand side of the above inequality can be rewritten as

\[
\frac{\text{load}^x(\tau)}{(1 - \delta_{\max}^x(\tau))} \leq m - (m - 1) \cdot \delta_{\max}^x(\tau)
\]

\[
\leq m \cdot (1 - \delta_{\max}^x(\tau)) + \delta_{\max}^x(\tau)
\]

Subtracting \( \delta_{\max}^x(\tau) \) from both sides yields

\[
\frac{\text{load}^x(\tau)}{(1 - \delta_{\max}^x(\tau))} - \delta_{\max}^x(\tau) \leq m \cdot (1 - \delta_{\max}^x(\tau))
\]

and rewriting the left-hand side of the above inequality yields

\[
\frac{\text{load}^x(\tau) - \delta_{\max}^x(\tau) \cdot (1 - \delta_{\max}^x(\tau))}{(1 - \delta_{\max}^x(\tau))} \leq m \cdot (1 - \delta_{\max}^x(\tau))
\]
Finally, dividing both sides by \((1 - \delta_{\text{max}}^x(\tau))\) yields
\[
m \geq \frac{\text{LOAD}^x(\tau) - \delta_{\text{max}}^x(\tau) \cdot (1 - \delta_{\text{max}}^x(\tau))}{(1 - \delta_{\text{max}}^x(\tau))^2}
\]

From Test 3.2, the function \(\text{cpu}^\text{EDF}_x(\tau)\) can therefore be written as
\[
\text{cpu}^\text{EDF}_x(\tau) \overset{\text{def}}{=} \min \{\text{BCL}_x(\tau), \text{BAK}_x(\tau), \text{BB}_x(\tau)\}
\]
where \(\text{BCL}_x(\tau), \text{BAK}_x(\tau)\) and \(\text{BB}_x(\tau)\) are respectively defined from the three conditions\(^1\) of Test 3.2, i.e.,
\[
\begin{align*}
\text{BCL}_x(\tau) & \overset{\text{def}}{=} \left\lceil \frac{\delta_{\text{diff}}^x(\tau) - \delta_{\text{max}}^x(\tau)}{1 - \delta_{\text{max}}^x(\tau)} \right\rceil \\
\text{BAK}_x(\tau) & \overset{\text{def}}{=} \max_{\tau_i \in \tau} \left\lceil \frac{\sum_{\tau_i \in \tau} \min\{1, \beta_i^x\} - \delta_k^x}{1 - \delta_k^x} \right\rceil \\
\text{BB}_x(\tau) & \overset{\text{def}}{=} \left\lceil \frac{\text{LOAD}^x(\tau) - \delta_{\text{max}}^x(\tau) \cdot (1 - \delta_{\text{max}}^x(\tau))}{(1 - \delta_{\text{max}}^x(\tau))^2} \right\rceil
\end{align*}
\]

### 3.3.5 Power dissipation model of the MPSoC platforms

For any given set of tasks \(\tau\) and any given global scheduling algorithm \(\mathcal{S}^x\), we formulate an upper-bound on the power dissipation of the MPSoC platform \(L_x\) executing the task set \(\tau\) as follows:
\[
Pwr^x(\tau) \overset{\text{def}}{=} \frac{U^x_{\text{sum}}(\tau)}{\text{cpu}_x^{\mathcal{S}^x}(\tau)} \cdot Pwr^x_{\text{run}} + \frac{\text{cpu}_x^{\mathcal{S}^x}(\tau) - U^x_{\text{sum}}(\tau)}{\text{cpu}_x^{\mathcal{S}^x}(\tau)} \cdot Pwr^x_{\text{idle}} (3.5)
\]
where the notations have the following interpretations:

1. \(Pwr^x_{\text{run}}\) denotes the power dissipation of an \(x\)-processor while running a task at maximal clock frequency. This power dissipation corresponds to the sum of the short circuit power dissipation \(Pwr_{\text{short}}\), the dynamic power dissipation \(Pwr_{\text{dyn}}\),

---

\(^1\)Notice that, for some infeasible real-time systems, one (or more) of these three functions can get negative. A negative value would distort the result from \(\text{cpu}^\text{EDF}_x(\tau)\). Hence, we consider that any of these three functions \(\text{BCL}_x(\tau), \text{BAK}_x(\tau)\) and \(\text{BB}_x(\tau)\) returns \(\infty\) if negative while computing the minimum value in \(\text{cpu}^\text{EDF}_x(\tau)\).
and the leakage power dissipation $P_{\text{leak}}$ (see Chapter 1 on page 49 for more details).

2. $P_{\text{idle}}^x$ denotes the power dissipation of an $x$-processor while it idles. Note that, although the processor is idling, this power dissipation is not only due to the leakage and subthreshold currents (see the power dissipation $P_{\text{leak}}$ introduced in Chapter 1, page 49). Indeed, an idling processor executes an infinite loop of NOP instructions (NOP = No OPeration) and thus, it dissipates a small amount of dynamic power. A common solution to avoid this dynamic power dissipation is to put the idling processors into a sleep mode. Recent processors usually provide several sleep modes, where each one differs from another in the amount of processor parts that it turns off, the leakage power that it dissipates while sleeping and the time (and energy) that it needs to return to the normal (i.e., operative) mode. Consequently, selecting the most appropriate sleep mode, in term of energy consumption, depends on the time during which the processor can sleep. This problem is not obvious and although selecting a suitable sleep mode can be a better option than idling a processor (the energy savings could be significant), we do not focus on such approaches in this research.

3. $\text{cpu}_{\tau}^x$ is the number of supplied processors in platform $L_\tau$. That is, the processors that are turned off are not counted in $\text{cpu}_{\tau}^x$.

4. $\frac{U_{\text{sum}}^x(\tau)}{\text{cpu}_{\tau}^x(\tau)}$ represents an upper-bound on the proportion of time during which the $\text{cpu}_{\tau}^x(\tau)$ supplied CPUs of $L_\tau$ will be running a task, considering an infinite interval of time.

Following the interpretation of $\frac{U_{\text{sum}}^x(\tau)}{\text{cpu}_{\tau}^x(\tau)}$, it holds that the expression

$$\frac{\text{cpu}_{\tau}^x(\tau) - U_{\text{sum}}^x(\tau)}{\text{cpu}_{\tau}^x(\tau)}$$

represents a lower-bound on the proportion of time during which the $\text{cpu}_{\tau}^x(\tau)$ supplied processors of $L_\tau$ idle. It results that $P_{\text{idle}}^\tau(\tau)$ is a long term upper-bound on the power dissipation of the MPSoC platform $L_\tau$ while executing the task set $\tau$. For a given DMP and a given task mapping $\{\tau_{\text{lp}}, \tau_{\text{hp}}\}$, an upper-bound on the power dissipation of the whole DMP is then given by:

$$P_{\text{wr}}(\tau_{\text{lp}}, \tau_{\text{hp}}) \overset{\text{def}}{=} P_{\text{wr}}^{\text{lp}}(\tau_{\text{lp}}) + P_{\text{wr}}^{\text{hp}}(\tau_{\text{hp}})$$

(3.6)
CHAPTER 3. OPTIMIZING THE HARDWARE DESIGN

3.3.6 Formulation of the problem

The problem of minimizing the energy consumption of a DMP while executing a set of sporadic constrained-deadline tasks can be stated as follows.

Let $\pi$ be any DMP composed of two identical MPSoC platforms $L_{lp}$ and $L_{hp}$ composed of $m_{lp}$ $lp$-processors and $m_{hp}$ $hp$-processors, respectively. Let $S^p$ (resp. $S^h$) be the global scheduling algorithms used by $L_{lp}$ (resp. $L_{hp}$), for which a sufficient schedulability test based on the number of required CPUs is known. For a given task mapping $\{\tau_{lp}, \tau_{hp}\}$, let $\text{crv}_{lp}^{S^p}(\tau_{lp})$ and $\text{crv}_{hp}^{S^h}(\tau_{hp})$ be the functions that return a sufficient number of $lp$-processors (resp. $hp$-processors) required to schedule the sets of tasks $\tau_{lp}$ (resp. $\tau_{hp}$) by $S^p$ (resp. $S^h$) without missing any deadline. Let $\tau$ be any set of $n$ sporadic constrained-deadline tasks. The goal of this study is to determine a task mapping $\{\tau_{lp}, \tau_{hp}\}$ (where $\tau_{lp} \cup \tau_{hp} = \tau$ and $\tau_{lp} \cap \tau_{hp} = \phi$) that satisfies the three following conditions:

1. $\text{crv}_{lp}^{S^p}(\tau_{lp}) \leq m_{lp}$
2. $\text{crv}_{hp}^{S^h}(\tau_{hp}) \leq m_{hp}$
3. $\text{Pwr}(\tau_{lp}, \tau_{hp})$ is as low as possible.

As mentioned above, determining a task mapping $\{\tau_{lp}, \tau_{hp}\}$ can be seen as a bin-packing problem. The two subsets $\tau_{lp}$ and $\tau_{hp}$ represent two bins. Each task $\tau_i$ represents an item whose the weight depends on the container ($\tau_{lp}$ or $\tau_{hp}$). The functions $\text{crv}_{lp}^{S^p}(\tau_{lp})$ and $\text{crv}_{hp}^{S^h}(\tau_{hp})$ give the capacity of each bin and $m_{lp}$ and $m_{hp}$ are their respective total capacity. However, in a classical bin-packing problem the capacity of each bin $B$ is given by the sum of the weights of the items placed into $B$. On the contrary, we assume here that the capacity of each bin is given by a complex function—the functions $\text{crv}_{x}^{S^x}(\tau)$—which is not linear in the weights of the items—the densities of the tasks—placed into the bins. Due to the complexity of these complex functions giving the capacities of the two bins, our problem is at least as complex as a classical bin-packing problem. This is the reason why we use heuristic algorithms to solve the problem formulated above, although we do not address its complexity in this thesis.

The optimal task mapping (or one of them) may be found by using an exhaustive search, but the worst-case computing complexity of such a search exponentially grows with the number of tasks in the application. As a result, we address in the following
section two popular heuristics for solving the problem described above with a reasonable time-complexity. Notice that all the task mapping methods that we propose are applied at system design-time.

3.3.7 Approximation algorithms

In this section, we denote a task mapping as a vector $V$ of $n$ elements, where $n$ is the number of tasks in the application $\tau$. The two sets $\tau^{lp}$ and $\tau^{hp}$ of the task mapping $V$ are henceforth defined as the set of tasks $\tau_i$ for which $V[i] = lp$ and $V[i] = hp$, respectively. We define an admissible task mapping as a task mapping for which:

1. $\exists i \in [1, n]$ such that $\delta_i^{lp} > 1$ and $V[i] = lp$
2. $\text{cpu}_{lp}^{\text{sqp}}(\tau^{lp}) \leq m_{lp}$
3. $\text{cpu}_{hp}^{\text{sqp}}(\tau^{hp}) \leq m_{hp}$

To approximate (one of) the optimal task mapping(s), we first consider the popular First-Fit Decreasing Density (FFDD) bin-packing heuristic. “Decreasing Density” means that tasks are considered (and then placed into a bin) by decreasing order of their density.

**FFDD.** Initially the set $\tau^{lp}$ is empty and the set $\tau^{hp}$ contains every task $\tau_i$ such that $\delta_i^{lp} > 1$. Then, every remaining task $\tau_j$ is placed in $\tau^{lp}$ if $\text{cpu}_{lp}^{\text{sqp}}(\tau^{lp} \cup \{\tau_j\}) \leq m_{lp}$, or otherwise in $\tau^{hp}$ if $\text{cpu}_{hp}^{\text{sqp}}(\tau^{hp} \cup \{\tau_j\}) \leq m_{hp}$. If a task can be placed in neither $\tau^{lp}$ nor $\tau^{hp}$, FFDD stops and returns the empty task mapping $\{\phi, \phi\}$.

Unfortunately, FFDD can be sometimes not sufficient to solve the problem formulated above. When it succeeds, it finds only one admissible task mapping and is therefore not suitable for a problem with a large range of solutions (i.e. when the number of tasks is high). Furthermore, we noticed during our numerous experiments (presented in Section 3.4 on page 226) that the partitions found by FFDD did not provide better results than partitions generated randomly. The reason stems from the observation explained above. In a classical bin-packing problem, the capacity of each bin $B$ is given by the sum of the weight of the items placed into $B$ and in the context of our problem, we assumed

---

1First-fit decreasing is a classical bin-packing algorithm and a tight upper-bound on its efficiency has been recently proposed in [16].
that the capacity of each bin is given by the function \( c \nu \tau \) which can be a complex function including different mathematical operators such as sums, products, max, min, etc. Thereby, our problem cannot be really considered as a classical bin-packing problem, because of this difference in the function that gives the capacity of each bin. We have also simulated other well-known strategies such as “Worst-Fit Decreasing”, “Best-Fit Decreasing”, “Next-Fit Decreasing”, etc. and we drew the same conclusions. This is why we consider in the following two more efficient heuristics: a Genetic Algorithm (GA) [24] and a Simulated Annealing (SA) [31].

3.3.7.1 Genetic Algorithm (GA)

Explaining how a genetic algorithm works is out of the scope of this thesis: the reader may consult [24] for details. Here, we only describe how we have parametrized this heuristic. Recall that \( \pi \) denotes the considered DMP and \( \pi \) is assumed to schedule the application \( \tau \) by the global scheduling algorithms \( S_{lp} \) and \( S_{hp} \) on \( L_{lp} \) and \( L_{hp} \), respectively.

**Initialization step.** During the initialization process, we randomly generated task mappings of \( \tau \) to form an initial population of 50 admissible task mappings. If the algorithm does not find any admissible task mapping after 50000 attempts, it returns the empty task mapping \( \{\phi, \phi\} \). If it finds less than 50 admissible task mappings after 50000 attempts, it returns the best one that it found and it stops. Otherwise, if the population size (50) is reached, the algorithm sequentially repeats the following steps\(^1\).

**Selection step.** Every task mapping \( \{\tau_{lp}, \tau_{hp}\} \) of the current population is rated by the function \( Pwr(\tau_{lp}, \tau_{hp}) \): the lower the task mapping rate, the higher its probability to be selected. The selection method that we used in our simulations is the popular and well-studied roulette wheel method [24].

**Reproduction step.** The goal of this step is to generate a next population (which is initially empty) of 50 admissible task mappings from the current population. This is achieved through the crossover operator. This operator takes as argument two task mappings \( V_1 \) and \( V_2 \) (selected in the current population through the roulette wheel selection method) and produces two new admissible task mappings \( v_1 \) and \( v_2 \).

\(^1\)At this stage, the “current population” is composed of 50 admissible task mappings and the “next population” is empty.
The crossover operator works as follows. First, a task index $k$ is randomly selected in the interval $[1, n - 1]$. Then, the crossover operator performs the following treatment:

\[
\forall i \in [1, k] : v_1[i] \leftarrow V_1[i] \text{ and } v_2[i] \leftarrow V_2[i]
\]

\[
\forall i \in [k + 1, n] : v_1[i] \leftarrow V_2[i] \text{ and } v_2[i] \leftarrow V_1[i]
\]

Figure 3.3: Illustration of the crossover operation. Basically, the value of $k$ divides the two “parent” solutions $V_1$ and $V_2$ into two pieces. The first “child” solution $v_1$ is composed of the first piece of $V_1$ and the second piece of $V_2$ whereas the second “child” $v_2$ is composed of the first piece of $V_2$ and the second piece of $V_1$.

The crossover operation is illustrated in Figure 3.3. While both $v_1$ and $v_2$ are not admissible, a new task index $k$ is randomly selected in $[1, n - 1]$ and the crossover operation is repeated. If after having considered all possible task indexes no admissible $v_1$ or $v_2$ (or both) have been successfully produced, then the crossover operator performs $v_1[i] \leftarrow V_1[i]$ (resp. $v_2[i] \leftarrow V_2[i]$) $\forall i \in [1, n]$. The two resulting admissible task mappings $v_1$ and $v_2$ are inserted in the next population, and the reproduction step is repeated until this next population reaches the appropriate size (50). This step ultimately results in a next population of admissible task mappings that are different from those in the current population\(^1\). Generally, the

\(^1\)Except if the crossover operations failed to produce two new task mappings $v_1$ and $v_2$ for all selected
rate of the task mappings in the next population is (in average) better than the one of the current population, since the best-rated task mappings are more likely to be selected for the crossover operation (thanks to the roulette wheel method).

**Termination step.** The selection and reproduction steps are repeated while the sum of the rates of all the task mappings of the current population is lower than that of the previous population. That is, populations are bred while the population rate decreases. Finally, the task mapping returned by this heuristic is the best-rated one found during the whole process.

During the whole process, all the generated populations are composed of only admissible task mappings. However, some implementations of GA allow to deal with non-admissible solutions, because passing through the non-admissible solution space sometimes allows to escape from local minima, or to find “shortcuts” to some better-rated admissible solutions. In the context of our problem, however, we know that the best task mapping $V_{\text{opt}}$ (in regard with the energy consumption) is $V_{\text{opt}}[i] = \lceil \text{lp } V_i \rceil$, which is often not admissible (otherwise, it could be found by FFDD). As a result, by allowing populations to contain non-admissible task mappings, we have noted that GA prematurely leaves the admissible solution space by generating populations in such a manner that their task mappings converge too quickly toward $V_{\text{opt}}$. That is, the produced task mappings enter the non-admissible solution space very quickly and never come back to the admissible solution space.

Since our GA is numerously invoked in our simulation presented in Section 3.4, all its parameters (i.e. the population size, etc) are set to relatively low values in order to limit the time-complexity of the simulations. For the same reason, we did not implement the “mutation” operator which is usually employed by the genetic algorithms (see [24] for details about this operator). Indeed, since only admissible task mappings are allowed in the populations, the mutation operator should ensure that the resulting task mapping will be still admissible. For some populations, it may be needed to consider numerous task mappings before finding one that guarantees this property. During our study, we noted that implementing this operator involves a consequent increase of the simulation time, while providing a negligible impact on the results.

pairs of task mapping $\{V_1, V_2\}$. However, this situation is highly unlikely and does not cause any failure of the algorithm.
3.3 Offline task mapping algorithms

3.3.7.2 Simulated Annealing (SA)

As for the genetic algorithm, it is out of the scope of this thesis to describe how a simulated annealing works, and we only present our parameters in this section. The interested reader may consult [31] for details about this algorithm. Again, as for our GA, our SA considers only admissible task mappings and the function that it tries to minimize is the function $Pwr(\tau^{lp}, \tau^{hp})$. Here is the list of the SA parameters that we used in our simulations.

**Initial task mapping.** During the initialization step, task mappings are randomly generated until finding an admissible one. If no admissible task mapping is found after 1000 attempts, the algorithm stops and returns the empty task mapping $\{\phi, \phi\}$.

**Neighborhood of a task mapping.** The neighborhood of a task mapping $V$ is the set of every admissible task mapping $V'$ such that

$$\exists j \in [1, n] \mid V'[j] \neq V[j]$$

and

$$\forall i \neq j : V'[i] = V[i]$$

Informally, an admissible task mapping $V'$ is a neighbor of $V$ if $V'$ differs from $V$ by one and only one task-platform assignment.

**Temperature.** The temperature $T$ is handled as follows: the initial temperature is set to 1 and at each iteration multiple of 100, the temperature is decreased such that $T \leftarrow 0.95 \times T$.

**Acceptance probabilistic function.** The acceptance probabilistic function $\text{Prob}(\Delta E, T)$ is defined as follows. $\Delta E$ is the difference between the rate of the current task mapping $V_{\text{cur}}$ and the rate of its selected neighbor $V_{\text{neighbor}}$, i.e.

$$\Delta E \overset{\text{def}}{=} Pwr(\tau^{lp}_{\text{neighbor}}, \tau^{hp}_{\text{neighbor}}) - Pwr(\tau^{lp}_{\text{cur}}, \tau^{hp}_{\text{cur}})$$

and $\text{Prob}(\Delta E, T) = e^{\frac{-\Delta E}{T}}$, where $T$ is the temperature at the current iteration. Notice that the probability of acceptance progressively decreases with the temperature.

**Termination condition.** The algorithm stops either after 1000 iterations or if all the task mappings in the neighborhood of the current one have been rejected. The
algorithm then returns the best-rated task mapping that it found during the whole process.

Notice that, for the same reasons as for our genetic algorithm, the parameters of our simulated annealing are chosen relatively low, and only admissible solutions are considered.

### 3.4 Simulation results

Since our methodology does not consider DVFS-capable processors, we do not compare the consumption of our proposed strategy with the consumption while using DVFS algorithms. Thereby, the goal of our simulations is to quantify the average energy savings provided by a DMP for a task set $\tau$ in regard with the consumption of an identical multi-processors platform $\pi^{\text{ident}}$ only composed of $\text{cpu}_{hp}^{\tau}(\tau)$ hp-processors. In order to consider environments where a DMP is justified, we shall always assume that the assignment of all the tasks to the platform $L_{lp}$ of the DMP is not admissible (for instance, because at least one task $\tau_i$ is such that $\delta_{i} > 1$).

The DMP that we consider in our simulations (noted $\pi^{\text{dmp}}$) is composed of 10 processors: 5 lp-processors and 5 hp-processors. The lp- and hp-processor cores are the Diamond 108 Mini and the Diamond 570T, respectively. The 108 Mini is an ultra-low power CPU with minimal gate count for lowest silicon cost whereas the core 570T is a extremely high-performance, 2- or 3-issue static superscalar processor [37]. We assume that the lp-processor cores run at 0.6V, are manufactured with a 130nm technology and are built with ARM’s Artisan Metro low-power standard-cell library. Due to the process technology, the standard-cell library, and the selected synthesis options, their clock speed are constrained to be well below 100MHz [29] and we assume that their clock speed is of 50MHz in our simulations. On the other hand, we assume that the cores 570T run at 1.2V, are manufactured with a 130nm technology and are built with ARM’s Artisan SageX standard-cell library. These processor cores can typically run at clock rates in excess of 200MHz [29] and we assume in our simulations that they run at 250MHz. Table 3.1 shows the characteristics of these processors from [29]. The idle/busy factor denotes the power dissipation ratio between a busy processor and an idle processor. Unfortunately, this factor is not given for the processors considered in this work and we assume that it is comprised between 4 and 11, based on other
processor characteristics [22]. We thus choose to set it to 8 in our simulations. Notice that Table 3.1 provides the required values for our processor model: the lp-processors can be modeled as \( P_{\text{run}}^{\text{lp}} = 0.21 + 50 \times 0.04 = 2.21, \ P_{\text{idle}}^{\text{lp}} = P_{\text{run}}^{\text{lp}} / 8 \approx 0.28 \) and the hp-processor as \( P_{\text{run}}^{\text{hp}} = 103.91, \ P_{\text{idle}}^{\text{hp}} \approx 12.99 \). The last line of Table 3.1 gives the speed of the processors, depending on their operating clock frequency. For a given worst-case execution time of a task expressed in number of instructions, the processor speeds allow to determine the worst-case execution time of the tasks upon both the lp- and hp-processors (denoted \( c_{i}^{\text{lp}} \) and \( c_{i}^{\text{hp}} \) in our task model).

<table>
<thead>
<tr>
<th>CPU characteristics</th>
<th>lp-processor</th>
<th>hp-processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating clock frequency (MHz)</td>
<td>50</td>
<td>250</td>
</tr>
<tr>
<td>Leakage power dissipation (mW)</td>
<td>0.21</td>
<td>1.41</td>
</tr>
<tr>
<td>Dynamic power dissipation (mW/MHz)</td>
<td>0.04</td>
<td>0.41</td>
</tr>
<tr>
<td>Idle/Busy factor</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Speed (MIPS/MHz)</td>
<td>1.34</td>
<td>1.59</td>
</tr>
</tbody>
</table>

**Table 3.1:** Description of the processor cores

Concerning the scheduling algorithms used by our \( \pi^{\text{dmp}} \), our simulations were first carried out while using Global-DM (Figure 3.4a) on both \( L_{\text{lp}} \) and \( L_{\text{hp}} \). On the other hand, \( \pi^{\text{ident}} \) also uses the scheduler Global-DM and is composed of only hp-processors (the same hp-processors than in \( \pi^{\text{dmp}} \)), but its number of processors is not statically fixed. For each generated real-time application \( \tau \), the number of hp-processors in \( \pi^{\text{ident}} \) is set to \( c_{\text{hp}}^{DM}(\tau) \) before computing its energy consumption while executing \( \tau \). We achieved this in order to compare our DMP \( \pi^{\text{dmp}} \) with an identical platform optimized in its number of available resources. In our second simulation (see Figure 3.4b), Global-EDF is used by \( L_{\text{lp}}, L_{\text{hp}} \) and \( \pi^{\text{ident}} \), and the number of processors of \( \pi^{\text{ident}} \) is therefore set to \( c_{\text{hp}}^{EDF}(\tau) \).

To study the energy savings provided by \( \pi^{\text{dmp}} \) over \( \pi^{\text{ident}} \) in the real-time context, we generate a large amount of sporadic constrained-deadline real-time applications. For each generated task set \( \tau \), a task mapping \( \{\tau^{\text{lp}}, \tau^{\text{hp}}\} \) is produced by jointly using FFDD, GA and SA algorithms. Then, the energy consumption of \( \pi^{\text{dmp}} \) while executing \( \{\tau^{\text{lp}}, \tau^{\text{hp}}\} \) is compared to the energy consumption of \( \pi^{\text{ident}} \) while executing \( \tau \). The energy consumption of \( \pi^{\text{dmp}} \) is determined thanks to Expression 3.6 and the energy consump-
CHAPTER 3. OPTIMIZING THE HARDWARE DESIGN

Figure 3.4: Our simulation results for both Global-DM and Global-EDF.
tion of $\pi^{\text{ident}}$ by Expression 3.5, page 218.

In our simulations, we studied how the tasks characteristics (i.e. the tasks density) affect the energy saved by the use of $\pi^{\text{dmp}}$ instead of $\pi^{\text{ident}}$. We distinguish between four groups of real-time tasks:

1. the HP tasks: $\delta_{\text{hp}}^i > 1$ and $\delta_{\text{hp}}^i \leq 1$. These tasks must be assigned to the platform $L_{\text{hp}}$ of the $\pi^{\text{dmp}}$.
2. the Heavy LP tasks: $1 \geq \delta_{\text{lp}}^i > 0.65$
3. the Middle LP tasks: $0.65 \geq \delta_{\text{lp}}^i \geq 0.3$
4. the Light LP tasks: $0.3 > \delta_{\text{lp}}^i > 0$

During our simulations, we generated a total of 12,000 task sets, each composed of 20 tasks. For each set, the first task is always generated in the HP group (in order to justify the use of DMP). Then, two task groups are selected and the 19 remaining tasks are generated in these two groups, where the number of tasks belonging to each of these two groups varies from 1 to 19. For instance, if the two selected groups are Heavy LP and Middle LP, our simulation process generates 2,000 task sets as follows.

1. First, it generates 100 task sets composed of 1 HP task (mandatory), 0 Heavy LP tasks and 19 Middle LP task.
2. Then, it generates 100 task sets composed of 1 HP, 1 Heavy LP and 18 Middle LP.
3. Then, 100 task sets of 1 HP, 2 Heavy LP and 17 Middle LP.
4. Then, 100 task sets of 1 HP, 3 Heavy LP and 16 Middle LP.
   ...
20. Finally, 100 task sets of 1 HP, 19 Heavy LP and 0 Middle LP.

This way of generating tasks is repeated for every pair of task groups, i.e, HP and Heavy LP, HP and Middle LP, HP and Light LP, Heavy LP and Middle LP, Heavy LP and Light LP, and Middle LP and Light LP, leading to a total of 12,000 generated task sets. This is carried out in order to study the impact of the task density over the energy saving provided by $\pi^{\text{dmp}}$. 
CHAPTER 3. OPTIMIZING THE HARDWARE DESIGN

Figures 3.4a and 3.4b show our results when the scheduling algorithms Global-DM and Global-EDF (respectively) are used on the platforms $L_{lp}$, $L_{hp}$ and $\pi^{ident}$. In both figures, the X-axis ranges the 6 pairs of selected tasks groups. For each of these 6 points in the X-axis, the Y-axis displays the number of tasks in the two selected groups. For instance, for $x = "MLP and LLP"$, $y = 14$ means that 14 tasks belong to the Light LP group, $(n - y - 1) = 5$ tasks belong to the Middle LP group and 1 task is the mandatory HP task. Finally, the Z-axis plots the energy saving provided by $\pi^{dmp}$ over $\pi^{ident}$. More precisely, for each couple of $(x, y)$ discrete values our simulator computes the energy saving between the execution of the 100 task sets upon $\pi^{dmp}$ (while determining an energy-optimized task mapping for each task set) and upon the platform $\pi^{ident}$ (while determining a sufficient number of hp-processors for each task set). The upper plane is the sum between the average energy saving and the standard deviation and the lower plane is the difference between the average energy saving and the standard deviation.

Notice that in both Figures 3.4a and 3.4b, there is no energy saving for the greater part of real-time systems containing many HP tasks. The reason is that the tasks generated in this space region have a so high workload that no (or too few) admissible task mapping is found for $\pi^{dmp}$. Since we obtain not enough information about the energy savings in order to provide significant statistics, these task sets are not taken into account and we set the energy saving to zero. On the other hand, we see that the average energy saving out of this region mainly vary between 20% and 40% for both global-DM and global-EDF.

3.5 How to handle mode changes?

At the beginning of this chapter, we mentioned that the proposed approaches can be applied during the execution of a multimode real-time system. The idea was to apply any of the proposed approach to each mode of the system, hence providing a task mapping $[\tau^{lp}, \tau^{hp}]$ for the task set of each mode. Together with each of these task mappings, our approaches determine two sufficient numbers $cru_{lp}^{sw}(\tau^{lp})$ and $cru_{hp}^{sw}(\tau^{hp})$ of lp- and hp-processors so that the tasks of $\tau^{lp}$ (resp. $\tau^{hp}$) of each mode can be successfully scheduled on the platform $L_{lp}$ (resp. $L_{hp}$) when only $cru_{lp}^{sw}(\tau^{lp})$ (resp. $cru_{hp}^{sw}(\tau^{hp})$) processors are powered on. In Figure 3.2 (page 213), we depicted a multimode real-time application composed of 5 operating modes, for which a task mapping is determined for each mode. That is, the set of tasks of each mode is split into two subsets $\tau^{lp}$ and $\tau^{hp}$ that
are scheduled upon its respective platform $L_{lp}$ or $L_{hp}$ using only a sufficient number of CPUs. The squares depicted in green are the supplied CPUs while the CPUs turned off are represented by crossed-out squares.

At first blush, a transition between two modes can seem problematic since two distinct modes can execute their own subsets $\tau^{lp}$ and $\tau^{hp}$ of tasks on two different numbers of processors. Therefore, particular rules have to be designed in order to decide the number of CPUs to turn on in the two platforms $L_{lp}$ and $L_{hp}$ so that all the deadlines are met during every transition phase from any mode $M'$ to any other mode $M$. Before going further in the design of these rules, let us introduce the following notations:

- $m_{lp}^{on}(M')$ and $m_{hp}^{on}(M')$ denote the number of processors powered on in the platforms $L_{lp}$ and $L_{hp}$ (respectively) during the execution of the mode $M'$.
- $\tau^{lp}(M')$ and $\tau^{hp}(M')$ denote the set of tasks that are executed on the platforms $L_{lp}$ and $L_{hp}$ (respectively) during the execution of the mode $M'$. 
- $m_{lp}^{on}(M', M')$ and $m_{hp}^{on}(M', M')$ denote the number of processors powered on in the platforms $L_{lp}$ and $L_{hp}$ (respectively) during every transition phase from mode $M'$ to mode $M'$.
- Concerning synchronous protocols (see Chapter 2, page 75 for the definitions related to transition protocols and multimode applications), it is easy to see that during every transition from any mode $M'$ to any other mode $M'$, no deadline can be missed provided

1. $m_{lp}^{on}(M', M') = m_{lp}^{on}(M')$ and
2. $m_{hp}^{on}(M', M') = m_{hp}^{on}(M')$.

At the end of the transition phases—when all the rem-jobs are completed in platform $L_x$ (where $x$ is again either $lp$ or $hp$)—all the new-mode tasks of $\tau^{x}(M')$ are enabled and $L_x$ switches its number of supplied processors to that of the new-mode, i.e., to $m_{x}^{on}(M')$. Formally, every transition from any mode $M'$ to any other mode $M'$ can be handle as follows:

1. Upon the MCR($j$), the rem-jobs issued from the task set $\tau^{x}(M')$ continue to be scheduled by $S^{i}$ (the old-mode scheduler) on the platform $L_x$ in which $m_{x}^{on}(M', M') = m_{x}^{on}(M')$ processors are powered on.
2. When all the rem-jobs are completed on platform $L_x$, $L_x$ switches its number of supplied processors to $m^{on}_x(M')$ and all the tasks of $\tau^i(M')$ are enabled. Then, these new-mode tasks are scheduled by $S^i$ (the new-mode scheduler) on $L_x$ in which $m^{on}_x(M')$ processors are powered on.

Figure 3.5: This illustration shows how mode transitions are handled by our synchronous protocol SM-MSO (described in Chapter 2, page 85), using the task mapping approach proposed in this chapter.

Figure 3.5 illustrates how a synchronous mode transition is managed according to the two rules described above. This figure depicts a mode transition between two modes $M^i$ and $M^j$. The top half represents the task mapping of each mode while the bottom half depicts the produced schedule for the mode transition $M^i \rightarrow M^j$ on platform $L_{hp}$. The task set $\tau^{hp}(M^i)$ is composed of 4 periodic tasks, all with the same period $t_1$ whereas the task set $\tau^{hp}(M^j)$ is composed of 3 tasks whose characteristics are not significant in this example. As illustrated in the top half of the picture, the four tasks of $\tau^{hp}(M^i)$ require the two processors of $L_{hp}$ to be powered on in order to be successfully scheduled.
3.5 How to handle mode changes?

whereas the three tasks of \( \tau^{hp}(M_i) \) require only 1 hp-processor. As specified by the first rule above, the two processors of \( L_{hp} \) are supplied during the mode transition since these two processors are powered on during the execution of the old-mode \( M_i \). At time 0, the system starts in mode \( M_i \) and each of the 4 tasks of \( \tau^{hp}(M_i) \) immediately generates its first job. At time \( t_1 \), these four tasks generate their second job. Then at time \( t_2 \), the system requests a mode change to the mode \( M_j \). According to the first rule above, the number of supplied processors does not change and the four rem-jobs continue to be scheduled by the old-mode scheduler upon the 2 hp-processors. At time \( t_3 \), all the rem-jobs are completed and the system switches to mode \( M_j \). That is, it loads the configuration of \( L_{hp} \) which is specified by the task mapping of \( M_j \) and one processor (here, \( \pi_2 \)) is therefore turned off since the task set \( \tau^{hp}(M_j) \) requires only one hp-processor in order to be successfully scheduled. All the tasks of \( \tau^{hp}(M_j) \) are then enabled and the system starts scheduling them by the new-mode scheduler. The schedulability of the application during the transitions is guaranteed thanks to Lemma 2.4, page 103, in which we showed that every rem-job meets its deadline while continuing to be scheduled by \( S^i \) during every transition from mode \( M_i \).

Concerning our asynchronous transition protocol AM-MSO, the following rule can be used:

1. Upon the MCR(\( j \)), the rem-jobs issued from the task set \( \tau^x(M_i) \) (\( x \) is either lp or hp) continue to be scheduled by \( S^i \) (the old-mode scheduler) on the platform \( L_x \) in which \( m^{on}_x(M_i, M_i) \) processors are powered on, where

\[
m^{on}_x(M_i, M_i) \overset{\text{def}}{=} \max\{m^{on}_x(M_i), m^{on}_x(M_j)\}
\]

2. When all the rem-jobs are completed on platform \( L_x \), \( L_x \) switches its number of supplied processors to \( m^{on}_x(M_j) \) and the new-mode tasks are scheduled by \( S^j \) (the new-mode scheduler) on \( L_x \).

The proof that this rule allows to meet all the job deadlines is given below. Basically, it is a direct consequence of some lemmas proved in Chapter 2.

**Lemma 3.1**

When a mode transition is requested from any mode \( M_i \) to any other mode \( M_j \), the schedulability is guaranteed for AM-MSO provided
CHAPTER 3. OPTIMIZING THE HARDWARE DESIGN

\[
\begin{align*}
m_{lp}^{on}(M^i, M^j) & \overset{\text{def}}{=} \max \{m_{lp}^{on}(M^i), m_{lp}^{on}(M^j)\} \\
m_{hp}^{on}(M^i, M^j) & \overset{\text{def}}{=} \max \{m_{hp}^{on}(M^i), m_{hp}^{on}(M^j)\}
\end{align*}
\]

Proof

Considering the notations introduced above, three situations can occur upon the MCR\((j)\) for both \(x = lp\) and \(x = hp\):

1. \(m_x^{on}(M^i) = m_x^{on}(M^j)\) or
2. \(m_x^{on}(M^i) < m_x^{on}(M^j)\) or
3. \(m_x^{on}(M^i) > m_x^{on}(M^j)\).

Case 1. The lemma obviously holds for the first case; since the number of supplied processors does not change, the validity of our asynchronous protocol has already been showed in Chapter 2.

Case 2. In this case, \(m_x^{on}(M^i, M^j) = m_x^{on}(M^j)\) and the schedulability of the rem-jobs is guaranteed thanks to

1. Lemma 2.4 (page 103) that shows that all the rem-jobs meet their deadline by being scheduled by \(S^i\) (the old-mode scheduler) on \(m_x^{on}(M^i)\) processors.
2. Lemma 2.7 (page 113). According to this lemma, any strongly work-conserving scheduler that is able to schedule a task set \(\tau\) upon an uniform platform \(\pi\) is also able to schedule \(\tau\) upon any uniform platform \(\pi^*\) such that \(\pi^* \supseteq \pi\). It is easy to see that the proof of this lemma can be adapted to any work-conserving scheduler (according to the definition given on page 36), any identical platform and any set of jobs (rather than a set of tasks). That is, the rem-jobs meet their deadline on \(m_x^{on}(M^j)\) processors since they meet their deadline on \(m_x^{on}(M^i)\) processors and \(m_x^{on}(M^i) > m_x^{on}(M^j)\) in this case.

Finally, the schedulability of the new-mode tasks is guaranteed since the task set \(\tau^i(M^j)\) is assumed to be schedulable on \(m_x^{on}(M^j)\) processors.
Case 3. In this case, $m_{\tau}^{on}(M^{i}, M^{j}) = m_{\tau}^{on}(M^{i})$ and the schedulability of the rem-jobs in the transition phase is guaranteed thanks to Lemma 2.4. Concerning the new-mode tasks of $\tau^i(M^i)$, we know by assumption that there are schedulable on $m_{\tau}^{on}(M^{i})$ processors. However in this case, the number $m_{\tau}^{on}(M^{i}, M^{j})$ of supplied processors during the transition phase is higher than $m_{\tau}^{on}(M^{i})$ and there can therefore exist one (or several) interval(s) of time during which the new-mode tasks profit from a number of processors higher than $m_{\tau}^{on}(M^{i})$. During this (these) time interval(s), we know from Lemma 2.7 (page 113) that every new-mode job meets its deadline. Finally, at the end of the transition phase, some new-mode jobs can have a lower remaining processing time than in the scenario in which only $m_{\tau}^{on}(M^{j})$ processors are available during the transition. According to the predictability showed in Lemma 2.2 (page 102), we know that all these new-mode jobs meet their deadline. The lemma follows.

3.6 Conclusion and future work

In this chapter, we have addressed the energy-aware scheduling problem upon a DMP, a particular platform model composed of two types of processors organized in two distinct identical platforms. We exhibited multiple advantages of such architectures from software and practical point of views. In particular we showed that our DMP architecture, in addition to be compatible with any processor architecture, considerably simplifies the energy-aware scheduling problem to a 2-bins packing problem for which an abundant literature is available. Besides, on the contrary to most of the existing DVFS solutions, our methodology is designed in such a manner that it focuses on the leakage consumption of the processors and is therefore more appropriate to the emerging processor technologies. Then, we explained why this new hardware platform and methodology is particularly well adapted for multimode application and finally, we showed in our simulations that, by taking as starting point some popular optimization heuristics, our methodology provides relevant energy saving upon a DMP architecture in regard to the consumption of a classical identical multiprocessor platform.
CHAPTER 3. OPTIMIZING THE HARDWARE DESIGN

References


REFERENCES


CHAPTER 3. OPTIMIZING THE HARDWARE DESIGN


CHAPTER 3. OPTIMIZING THE HARDWARE DESIGN

Asia and South Pacific Design Automation Conference (ASP-DAC ’00), pages 147–152, New York, NY, USA, 2000. ACM.


Reducing the energy consumption by using the DVFS framework

« Si chaque découverte entraîne plus de questions qu’elle n’en résout,
la part d’inconnu s’accroît à mesure que nos connaissances augmentent. »
Pierre Vathomme

Contents

4.1 Introduction ...................................................... 245
4.2 Model of computation and assumptions ...................... 246
4.3 Related work ..................................................... 250
4.4 Contribution and organization of the chapter ............... 258
4.5 Offline speed determination ................................... 259
4.6 Online slack reclaiming algorithms .......................... 277
4.7 Simulation results ................................................. 314
4.8 Conclusion ....................................................... 331
CHAPTER 4. EXPLOITING THE DVFS FRAMEWORK

Abstract

In this chapter, we address the energy-aware scheduling problem of sporadic and constrained-deadline tasks. We consider identical multiprocessor platforms composed of DVFS-capable processors only. That is, we exploit the dynamic voltage and frequency scaling feature of the processors in order to reduce their consumption. We propose several distinct algorithms that reduce the consumption of the processors while scheduling the tasks. First, we present two offline methods in the sense that they are invoked at system design-time. The first one assigns the same level of voltage/frequency to every processor and all of them maintain this operating configuration constant at run-time. The second method assigns a constant level of voltage/frequency to each processor with the interpretation that two processors can operate at different voltage/frequency at run-time, but none of them can modify its assigned configuration. Then, we propose two online strategies with the interpretation that these approaches can modify the voltage and frequency of any processor at run-time. The first one is called MORA for “Multiprocessor Online Reclaiming Algorithm”. It is a slack reclamation scheme that detects early task completions and slows down the processors speeds in such a manner that the resulting speeds match the current workload as closely as possible. Our second online strategy is called MOTE for “Multiprocessor One Task Extension”. This technique anticipates the intervals of time during which the processors will idle and adapt the current processors speeds accordingly. We demonstrate that both MORA and MOTE do not jeopardize the schedulability of the application and we show how these techniques can be combined together. Finally, we perform simulations showing that all these methods can significantly reduce the energy consumption of the processors.

1Slowing down the speed of processor reduces to lower its operating voltage and frequency.
4.1 Introduction

Amongst the hardware and software techniques aimed at reducing energy consumption, supply voltage reduction is particularly effective. The reason is that the power dissipated in CMOS circuits is grossly proportional to the square of the supply voltage (see Section 1.4.3, page 44 for details about the consumption of the processors). Obviously, reducing the supply voltage cannot be achieved without reducing the clock frequency of the processor, involving a diminution of the processor speed. Nowadays, numerous processors offer several levels of supply voltages (each associated to a specific frequency) and most of them can modify at run-time their selected level of voltage/frequency. Therefore, many researches have been conducted in the field of Dynamic Voltage and Frequency Scaling (DVFS). When real-time constraints are a matter of concern, the extent to which the processor speed can be reduced depends on the characteristics of the tasks (execution time, release time, deadline, etc.) and on the underlying scheduling algorithm.

Currently, many modern processors can operate at various supply voltages, where different supply voltages lead to different clock frequencies and to different processing speeds. These processors are referred to as DVFS-processors hereafter, and among popular such processors, one can cite the Intel PXA27x processor family (see [43] for a complete description of this processor technology), used by many PDA devices (a non-exhaustive list of such devices can be found in the website [41]). Many modern computer systems, and especially embedded systems, are now equipped with such DVFS-processors that allow voltage and frequency variations, and these systems adopt various energy-efficient strategies for managing their applications intelligently by profiting from this DVFS feature. To understand the relation between the working voltage/frequency of a processor and its power dissipation, we refer the reader to Section 1.4.3 on page 44. In addition to this recent DVFS-capability, many modern embedded systems are now built upon multiprocessor platforms, mainly because of their high-computational requirements. As pointed out in [3, 4], another advantage is that multiprocessor systems are more energy efficient than equally powerful uniprocessor platforms, because raising the frequency of a single processor results in a multiplicative increase of the consumption while adding processors leads to an additive increase. As a result, the application of the DVFS framework on multiprocessor platforms became one of the most studied research fields in the concern of low-power system design.
CHAPTER 4. EXPLOITING THE DVFS FRAMEWORK

4.2 Model of computation and assumptions

4.2.1 Platform model

Unlike the work considered in [3], this chapter addresses energy-aware scheduling problems where the number of processors in the platform is assumed to be known beforehand. This constraint can be imposed by the availability of hardware components or by design considerations not related to the energy consumption. Also, the task characteristics are often not precisely established during the design of the hardware and the hardware architecture is usually designed so that it offers a computing performance higher than that required by the application. In this chapter, we consider multiprocessor platforms composed of a known and fixed number \( m \) of DVFS-identical processors \( \{\pi_1, \pi_2, \ldots, \pi_m\} \), according to the following definition.

**Definition 4.1 (DVFS-identical processors)**

Processors are said to be “DVFS-identical” if and only if the three following properties are satisfied:

1. all the processors of the platform make available the same levels of couples \(<voltage, frequency>\),
2. every processor is able to change at run-time its current operating level of \(<voltage, frequency>\),
3. two processors operating at a same level of \(<voltage, frequency>\) consume the same amount of energy and execute the same amount of execution units over time.

In addition, we also consider that the processors are independent according to the interpretation given in [53, 65] and repeated below.

**Definition 4.2 (dependent and independent DVFS-processors)**

DVFS-processors are said to be independent if at any instant during the execution of the application, two distinct processors can operate at two distinct levels of \(<voltage/frequency>\). Otherwise, the processors are said to be dependent.
4.2 Model of computation and assumptions

In order to save energy, the main idea behind energy-efficient DVFS algorithms is to adapt the operating level of \(<\text{voltage, frequency}>\) of the processor(s) to its (their) activity. This can be achieved either at design-time or at run-time, or both. However, from a software point of view, if DVFS algorithms managed directly the voltage and frequency of the processors, they would be very sensitive to the hardware characteristics. Indeed, each family of DVFS-processors provides its own set of available levels of \(<\text{voltage, frequency}>\) and thus, the compatibility of the DVFS algorithms with other hardware designs would suffer from a serious inflexibility if they dealt directly with the voltage and the frequency. To avoid this undesired hardware-sensitivity, most energy-aware algorithms use the notion of “processor speed” rather than managing the \(<\text{voltage, frequency}>\) levels. Formally, the speed \(s\) of a processor is defined as the ratio of its operating frequency \(f\) over its maximum frequency \(f_{\text{max}}\), i.e., \(s \overset{\text{def}}{=} \frac{f}{f_{\text{max}}}\). According to this definition, the maximum processor speed \(s^{\text{max}}\) is \(\frac{f_{\text{max}}}{f_{\text{max}}} = 1\) whatever the considered processor technology. That is, DVFS algorithms return only processor speed(s) and we assume that these speeds are then converted by the OS into a level of \(<\text{voltage, frequency}>\) available to the processor(s).

As mentioned earlier, any practical processor has only a finite number \(K\) of available levels of \(<\text{voltage, frequency}>\), thus providing only a finite number of available speeds. Without loss of generality, we assume that the available frequencies are ordered by increasing order and are denoted by \(f_1, f_2, \ldots, f_K\); Their corresponding speeds are denoted by \(s^1, s^2, \ldots, s^K\) (respectively), where \(s^1\) and \(s^K\) will be denoted by \(s^{\text{min}}\) and \(s^{\text{max}}\), respectively. Additionally, we denote by \(\text{Pwr}_{\text{run}}(s^i)\) (for \(i = 1, 2, \ldots, K\)) the relative power dissipation while the processor is running at speed \(s^i\), i.e., \(\text{Pwr}_{\text{run}}(s^{\text{max}})\) is assumed to be 100% and \(\text{Pwr}_{\text{run}}(s^i)\) (with \(1 \leq i < K\)) is expressed as a fraction of \(\text{Pwr}_{\text{run}}(s^{\text{max}})\). In the Appendix (Section 4.8 on page 347), we have outlined the available levels of \(<\text{voltage, frequency}>\) and the corresponding relative power dissipation of several processors commonly used in the practical world. These characteristics will be used later in our experiments in Section 4.7 (page 314).

This chapter focuses on providing a theoretical framework that can be then adapted to any practical situation. With this intention, all the methods proposed in this chapter make the following two assumptions:

1. The operating speed of any processor can be set to any value within \([s^{\text{min}}, s^{\text{max}}]\).
   Even though this assumption is highly unrealistic, it has been widely used in the
literature because the results obtained under this assumption can easily be adapted to practical processors with a limited number of available frequencies. Indeed, if the computed speed \( s \) does not belong to the set of available speeds provided by the processors, the available speed \( \bar{s} \) directly higher than \( s \) can always be selected instead.

2. Any job that executes on any processor running at speed \( s \) for \( R \) time units completes \( s \times R \) execution units. Again, this assumption is very pessimistic since it considers that every instruction of the executed code brings only the processor into play and its execution time depends therefore only on the processor speed. Obviously, this assumption is not always true. For instance, the execution time of a simple piece of code that reads a data from a file depends not only on the speed of the processor, but also on the speed of the memory, the current traffic on the communication bus, etc.

4.2.2 Application model

This chapter focuses on the schedule of only one operating mode during the execution of a multimode real-time application. That is, in the scope of this chapter, the application \( \tau \) is seen as a set of \( n \) functionalities denoted by \( \{\tau_1, \tau_2, \ldots, \tau_n\} \). Every functionality \( \tau_i \) is modeled by a sporadic and constrained-deadline task characterized by three parameters \( \langle C_i, D_i, T_i \rangle \) with the same interpretations as those given in Chapter 1 (page 12), except that the WCET \( C_i \) of every task \( \tau_i \) is assumed to be given at maximal processor speed \( s_{\text{max}} \). According to our definition of the speed, a processor running at speed \( s_{\text{max}} = 1 \) may take up to \( C_i \) time units to complete a job \( \tau_{i,j} \) and, at a given speed \( s \), its WCET is \( \frac{C_i}{s} \). Figure 4.1 depicts an example of a sporadic execution pattern of a single task \( \tau_1 = \langle 2, 4, 5 \rangle \) running on a single processor \( \pi_1 \). Its first and second jobs are executed at maximal speed \( s_{\text{max}} \) and its third job is executed at speed \( \frac{1}{2} \).

The density \( \delta_i \) of the task \( \tau_i \) is also assumed to be given at maximal processor speed \( s_{\text{max}} \), i.e., \( \delta_i \overset{\text{def}}{=} \frac{C_i}{T_i} \). Recall that the maximal density \( \delta_{\text{max}}(\tau) \) of any application \( \tau \) is defined as \( \delta_{\text{max}}(\tau) \overset{\text{def}}{=} \max_{i=1}^{n} \{\delta_i\} \) and its generalized density is defined as \( \delta_{\text{sum}}(\tau) \overset{\text{def}}{=} \sum_{i=1}^{n} \delta_i \). Finally, the problems and solutions presented in this chapter are addressed under the following assumptions:

1. Job migrations are permitted and are carried out with no loss or penalty.
4.2 Model of computation and assumptions

Figure 4.1: Schedule of $\tau_1 = (2, 4, 5)$, where $\tau_i = (\text{WCET}, \text{deadline}, \text{period})$. According to our definition of the speed of a processor, the execution time of job $\tau_{1,3}$ is twice as long as that of $\tau_{1,1}$ and $\tau_{1,2}$ since its execution speed is twice as slow.

2. Job parallelism is forbidden, i.e., jobs can execute on at most one processor at any instant in time.

3. The density of every task is not larger than 1, since a task with a density larger than 1 is never able to meet its deadlines if it executes for its WCET (because we assume that task parallelism is forbidden).

4. All the tasks are independent, i.e., there is no communication, no precedence constraint and no shared resource (except the processors) between them.

5. Speed transitions are performed without any time or energy overhead.

4.2.3 Scheduling specifications

This chapter considers global, preemptive, work-conserving and FJP scheduling algorithms, according to the definitions given in Chapter 1, Section 1.3.5 (on page 29). Global Deadline Monotonic and Global Earliest Deadline First (EDF) [9] are just some examples of such scheduling algorithms. We distinguish between offline and online energy-aware algorithms according to the following definition.
CHAPTER 4. EXPLOITING THE DVFS FRAMEWORK

Definition 4.3

An energy-aware algorithm is said to be offline if it determines the speed(s) of the processor(s) at system design-time, i.e., before the execution of the application. Otherwise, the algorithm is said to be online.

4.3 Related work

The problem of scheduling applications to meet all deadlines while reducing the processor voltage (and frequency) as much as possible was first addressed on uniprocessor platforms. Nowadays, there is an abundant literature covering numerous task and platform models, including multiprocessor platform models. The following two sections summarize the most interesting studies in the uniprocessor and multiprocessor context, respectively.

4.3.1 Uniprocessor energy-aware algorithms

The state-of-art that we present here is not exhaustive. Below, we cite some of the most interesting studies (in our opinion) in this research field.

In 1995 Yao et al. proposed in [71] an algorithm that saves energy while scheduling a finite set of jobs within a given interval of time. This algorithm assigns a constant speed to each job and its time-complexity was proved to be $O(n^2)$, where $n$ is the number of jobs\(^1\). Although the authors claim that the time-complexity can be reduced to $O(n \log(n))$ by using elaborate data structures such as “segment trees”, they do not provide any further details about this improvement.

In 1998 Hong et al. proposed in [40] several simple online heuristics for the problem of scheduling both periodic and aperiodic tasks. These heuristics are based on EDF and the main idea behind them is to adjust the processor speed at each job release according to the current workload. In short, upon each job release, their heuristics determine to what extent the speed of the processor can be reduced so that every currently active job is guaranteed to meet its deadline. Sporadic

\(^1\)Quan and Hu argue in [60] that the complexity is $O(n^3)$ because the algorithm requires to compute a “critical interval” for each job and the complexity for computing such an interval is $O(n^2)$. Therefore, $n$ such computations (one for each job) lead to a complexity of $O(n^3)$. We agree with the analysis provided in [60].
job releases are subject to acceptance tests that decide whether the released job may be scheduled without causing any periodic task (and/or previously accepted sporadic job) to miss its deadline using the highest speed of the processor. The same year, Ishihara and Yasuura [45] presented a model of dynamically variable voltage processor and basic theorems for power-delay optimization. In the same paper, they also formulated the static voltage scheduling problem as an integer linear programming (ILP) problem.

**In 1999** Shin and Choi presented in [62] a simple online strategy that reduces energy consumption for priority-driven FTP schedulers.

**In 2000 and 2001** Mossé et al. [54] and Shin et al. [61] proposed some solutions based on the notion of compiler-assisted scheduling. Here, each task $\tau_i$ is divided into sections $s_{1i}, s_{2i}, \ldots, s_{ki}$ for which the WCET is known. The main idea of these solutions is to re-compute the processor speed at the end of each section $s_j$ according to the difference between the cumulated WCET of sections $s_{1i}, s_{2i}, \ldots, s_{ji}$ and the time which was actually needed to reach the end of section $s_j$. In [6], Aydin et al. provide an efficient solution for periodic real-time tasks with (potentially) different power consumption characteristics. They show that a periodic task $\tau_i$ can run at a constant speed $s_i$ at every instance without hurting optimality and they design an $O(n^2 \cdot \log n)$ algorithm to compute the optimal $s_i$ values. Additionally, they also prove that the EDF scheduling policy can be used to obtain a feasible schedule with these optimal speed values. Finally, one can cite the work of Quan and Hu [60] that proposes a DVFS algorithm more efficient (but more complex as well) than that proposed in 1999 by Shin and Choi [62].

**In 2001 and 2002** Lorch and Smith [51] and Gruian [34, 35] proposed a new class of algorithms known as stochastic scheduling. The main idea behind stochastic scheduling is to start the execution of every job at a low speed and to gradually increase (if needed) its execution speed so that it can meet its deadline even in the worst-case scenario—the scenario in which every job executes for its WCET. Basically, since the jobs usually complete without consuming their WCET, higher speeds are rarely used and hence energy is not wasted. These papers assume, however, that the distributions of the job execution times are known at system design-time and the way following which the execution speed of a job is modified over time is determined thanks to these distributions. The same years, another interesting paradigm is addressed by Aydin et al. [5, 7], Pillai et al. [58] and Zhang et al. [75] who proposed different slack reclaiming strategies. Such strategies dynamically
collect the “unused” computation time at the end of each job, i.e., the difference between the WCET of the job and its actual execution time, and share this unused time among the remaining active jobs.

**In 2003** Yun and Kim proved in [73] that computing the optimal voltage schedule of a set of tasks under FTP strategies is NP-Hard and they present an approximation scheme running in polynomial time that gives a schedule as close to optimal as desired. The same year, Irani et. al [44] examined two different mechanisms for saving power in battery-operated embedded systems. The first one consists in adjusting the processor speed at run-time while the second one places the processor in a sleep state if it idles. Besides, they take into account the fact that a fixed amount of energy is required to bring the system back into an active state in which it can resume work. To the best of our knowledge, this is the first theoretical analysis of systems that can use both mechanisms, i.e., DVFS feature and processor sleep modes. In [50], the authors investigated scheduling policies to reduce leakage consumption in real-time systems. They showed that the overall leakage consumption can be reduced by an order of magnitude using simple scheduling techniques.

**In 2004** Bansal et al. [13] provided a tight upper-bound on the competitive ratio (in term of consumption) of some previous algorithms. Then in the same paper, they turned to the problem of dynamic speed scaling to minimize the maximum temperature that the device ever reaches, again subject to the constraint that all jobs finish by their deadlines. In [48], the authors focused on the leakage currents. More precisely, they pointed out the fact that, while dynamic voltage and frequency scaling is known to reduce the dynamic consumption, reducing the speed of the processors also causes increased leakage energy drain by lengthening the interval of time over which a computation is carried out. Therefore, for minimization of the total energy consumption, they established the need for determining an operating point that they called the critical speed. Based on this critical speed, they computed processor slowdown factors and showed by simulation experiments that the critical speed slowdown results in up to 18% energy gains over a leakage oblivious dynamic voltage scaling.

**In 2005** Gaujal et al. proposed in [32] the “Shortest-path” algorithm that schedule FIFO tasks without missing any deadline and while considerably reducing the consumption. This study reduces the scheduling problem to the problem of finding a shortest path in a 2-dimension space. In [76], Zhang et al. presented an
optimal procrastinating voltage and frequency scheduling (OP-DVS) for hard real-time systems using stochastic workload information. Algorithms are presented for both single-task and multi-task workloads. Offline calculations provide real-time guarantees for worst-case execution, and online scheduling reclaims slack time and schedules tasks accordingly. The OP-DVS algorithm is provably optimal in terms of energy minimization with no deadline misses and simulation results show up to 30% energy savings for single-task workloads and 74% for multi-task workloads compared to using a constant worst-case execution voltage. In [68], Xu et al. studied the problem of minimizing energy consumption in real-time embedded systems that execute variable workloads on uniprocessor platforms. This problem is about how to decide tasks’ running speeds (speed schedule) before they are scheduled to execute. They considered the frame-based task model in which all the tasks share a common period (with deadlines equal to this period) and this “frame” is indefinitely repeated. For this task model, they showed that it is possible to incorporate the dynamic behavior of the tasks into the speed schedule to, along with the dynamic slack reclamation technique, minimize the expected (total) energy consumption of the system.

In 2007 Xu et al. [67] dealt with energy-aware real-time system scheduling DVFS for energy-constrained embedded systems that execute variable and unpredictable workloads. They considered the frame-based task model and provided a practical approach for obtaining optimal (or provably close to optimal) stochastic inter-task, intra-task, and hybrid DVFS schemes under realistic power models in which the processor only provides a set of discrete speeds, no assumption is made on power/frequency relation, and speed change overhead is considered. Then in [69], the same authors investigated DVFS schemes that aim at minimizing expected energy consumption for frame-based hard real-time systems, where the probability distribution of the execution time of each task is assumed to be known beforehand. Their investigation considers various DVFS strategies (i.e., intra-task DVFS, inter-task DVFS, and hybrid DVFS) and both an ideal system model (i.e., assuming unrestricted continuous frequency, well-defined power/frequency relation, and no speed change overhead) and a realistic system model (i.e., the processor provides a set of discrete speeds, no assumption is made on power-frequency relation, and speed change overhead is considered). The highlights of the investigation are two practical DVFS schemes: Practical PACE (PPACE) for a single task and Practical Inter-Task DVFS (PITDVS2) for multi-task frame-based applications.
CHAPTER 4. EXPLOITING THE DVFS FRAMEWORK

In 2008, Berten et al. considered schedulability conditions for stochastic real-time tasks. First, they presented a schedulability condition for frame-based stochastic real-time tasks and also examined several algorithms to check the schedulability of a given strategy. Second, they proposed an approach based on the schedulability condition to adapt a continuous-speed-based method to a discrete-speed system. The approach is able to stay as close as possible to the continuous-speed-based method, but still guaranteeing the schedulability. It is showed by simulations that the energy saving can be more than 20% for some system configurations.

From 2008 to present, much less studies have been conducted in the uniprocessor context, because it is undeniable that today technology evolves to multiprocessor/multi-core architectures. Some of the most interesting researches (still in our opinion) in the multiprocessor context are cited in the following section.

4.3.2 Multiprocessor energy-aware algorithms

In 2007, Chen et al. [26] proposed a state-of-the-art review about energy-aware algorithms in multiprocessor environments. As mentioned in this state-of-the-art, many studies achieved between years 2003 and 2006 consider the frame-based task model. Among the most interesting studies with this task model, one can cite the following ones. In [77], Zhu et al. explored online slack reclamation schemes for dependent and independent frame-based tasks. In [27], Kuo et al. addressed independent tasks (where task migrations are not allowed) and proposed a set of energy-efficient scheduling algorithms with different task remapping and slack reclamation schemes. In [24], the authors provided some techniques with and without allowing task migration, where all the tasks are assumed to consume a same amount of energy in any given interval of time and each processor may run at its own selected speed, independently from the speeds of the other processors. In [23], the authors considered that the tasks might have different energy consumption profiles, i.e., two tasks running on a same processor do not necessarily consume a same amount of energy. Chen et al. [25] targeted energy-efficient scheduling of periodic real-time tasks over multiple DVFS processors with the considerations of leakage power dissipation due to leakage currents. Bautista et al. [19] focused on soft real-time applications and proposed an energy-aware real-time scheduler for a state-of-the-art multicore multithreaded processor, which implements dynamic voltage scaling techniques. They evaluated different scheduling alternatives.
4.3 Related work

and showed through experimental results that using a fair scheduling policy, their proposed algorithm provides, on average, energy savings ranging from 34% to 74%. In [70], energy-aware multiprocessor scheduling of frame-based tasks was explored for multiprocessor architectures, in which all the processors must share the same speed at any time. More recently, in 2009, Vandy Berten and Joël Goossens proposed in [21] a slack reclamation scheme for identical multiprocessor platforms, while considering frame-based tasks whose distribution of the execution times is assumed to be known at system design-time.

Targeting a sporadic task model, Anderson and Baruah [2] explored in 2008 the trade-off between the total energy consumption of task executions and the number of required processors, where all the tasks run at the same common speed. The same year, we addressed in [57] the energy-aware scheduling problem of sporadic constrained-deadline tasks using dynamic voltage scaling upon identical multiprocessor platforms. We proposed two distinct algorithms: an offline speed determination mechanism that provides a same speed for every processor (this speed guarantees that all deadlines are met if the jobs are scheduled using EDF) and an online and adaptive speed adjustment mechanism that reduces the energy consumption while executing the task set. This second method is called MOTE and is described in details in Section 4.6.3, page 297. Then in 2009, we presented in [55] the algorithm MORA. This algorithm also considers sporadic constrained-deadline tasks and identical multiprocessor platforms. To the best of our knowledge, MORA is the first online “slack reclaiming” algorithm that targets sporadic tasks and multiprocessor platforms. Furthermore, MORA takes into account the fact that two distinct tasks may have distinct consumption profiles, i.e., two tasks running on two identical processors for a same amount of time do not necessarily consume the same amount of energy. MORA can be combined with MOTE, which leads to a considerable reduction of the processor energy consumption. A complete description of this algorithm MORA is given in Section 4.6.2, page 281.

As introduced in Chapter 3 (Section 3.1.1, page 201), the leakage consumption has become more and more significant over the years and, because the dynamic consumption becomes less important as the leakage consumption increases, the energy savings provided by DVFS algorithms become less and less significant. For this reason, the real-time community currently shows a tendency to investigate more and more the system-wide energy optimization problem (although this problem has already been addressed
CHAPTER 4. EXPLOITING THE DVFS FRAMEWORK

in former researches such as the one presented in [47] for instance). The system-wide energy optimization problem consists in reducing the energy consumption of the whole system rather than focusing only on the consumption of the processors. In our opinion, a promising path for future researches is presented in [74] where the authors propose a realistic DVFS energy model that considers CPU, system bus, memory and task set characteristics at multiple frequency settings. Also, a particular interest is currently showed in the DPM (Dynamic Power Management) framework. DPM is used to reduce energy consumption of off-chip devices by transitioning the processors from the active to a sleep state. One of the most recent studies in this research field has been achieved this year (2010) by Kong et al. [49]. In that paper, the authors adopt the frame-based task model and develop optimization algorithms based on 0-1 Integer Non-Linear Programming for different system configurations.

4.3.3 Energy-aware algorithms with other concerns

In the literature, many papers focus on the energy consumption of the system without being limited to the energy consumption of the processors. For instance, Cheng et al. [28] analyze the problem of online energy-aware I/O scheduling for hard real-time systems based on the preemptive periodic task model with non-preemptible shared resources. That is, they focus on power conservation for I/O devices, rather than on the processors power dissipation. In their paper, they propose an online energy-aware I/O scheduling algorithm: Energy Efficient Device Scheduling with Non-preemptible Resources (EEDS NR). The EEDS NR algorithm performs device power state transitions to save energy, without jeopardizing temporal correctness, and they provide an evaluation of the approach which shows that it yields significant energy savings.

Other papers are concerned with the energy consumption of the processors but they also consider other constraints such as temperature and quality-of-service (QoS) requirements. For instance, the authors of [13] address the problem of dynamic speed scaling to minimize the maximum temperature that the device ever reaches, subject to the constraint that all jobs finish by their deadlines. They assume that the device cools according to Fourier’s law and show how to solve this problem in polynomial time, within any error bound. In [66], Xu et al. address the problem of energy-aware scheduling of streaming applications, which are represented by task graphs, on multiprocessor platforms using on/off and dynamic voltage and frequency scaling on a per-processor basis. Their goal is to minimize the energy consumption of streaming applications while
4.3 Related work

satisfying two typical QoS requirements, namely, throughput and response time. To the best of our knowledge, this paper was the first work to tackle this problem. They made a key observation: the trade-off between leakage power and dynamic power play a critical role in both parallel processing and pipelining that are used to reduce energy consumption in the scheduling process. Based on this observation, they proposed two scheduling algorithms, Scheduling1D and Scheduling2D, for linear and general task graphs, respectively.

An abundant literature is also available about the problem of reducing the consumption of wireless sensor networks. Focusing on real-time systems, one can cite [64] for instance, where the authors address the scheduling problem of periodic implicit-deadline tasks upon a networked real-time embedded system in which each node supports both dynamic voltage and frequency scaling. Each task consist of several sub-tasks exchanging messages where each sub-task has precedence constraints with other sub-tasks. They present static (centralized) and dynamic (distributed) energy gain based slack allocation algorithms which reduce the total energy consumption, while guaranteeing the ready time, deadline and precedence constraints.

Some papers also address more exotic platforms such as [42] or [72] for instance. In [42], Hung et al. address the energy-efficient real-time scheduling problem in systems equipped with a DVS processor and a non-DVS processing element (PE). They consider task scheduling under different energy consumption models of the non-DVS PE. When the energy consumption of the non-DVS PE is independent on the assigned workload, they developed a fully polynomial-time approximation scheme for energy-efficient scheduling. When the energy consumption of the non-DVS PE depends on the assigned workload, they developed a 0.5-approximation algorithm to maximize the energy saving, compared to the execution of tasks on a DVS processor. In [72], the authors focus on both power and performance of enterprise data centers. In traditional solutions, the energy consumption of a server is reduced by transitioning hardware components to lower-power states. Unfortunately, these approaches cannot be directly applied to today’s data centers that rely on virtualization technologies. Indeed, virtual machines running on the same physical server are correlated, because the state transition of any hardware component will affect the application performance of all the virtual machines. As a result, reducing power solely based on the performance level of one virtual machine may cause another to violate its performance specification. In
such a context, the authors propose a two-layers control architecture based on well-established control theory. The primary control loop adopts a multi-input-multi-output control approach to maintain load balancing among all virtual machines so that they can have approximately the same performance level relative to their allowed peak values. Then, the secondary performance control loop manipulates CPU frequency for power efficiency based on the uniform performance level achieved by the primary loop. Their paper provides empirical results that demonstrate that their control solution can effectively reduce server energy consumption while achieving required application-level performance for virtualized enterprise servers.

4.4 Contribution and organization of the chapter

The following sections address the problem of minimizing the energy consumed while executing a single set of sporadic and constrained-deadline tasks upon a fixed number of identical processors. The scheduler is assumed to be global, preemptive, work-conserving and FJP, according to the definitions introduced in Chapter 1, Section 1.3.5 (page 29).

In short, this chapter does not focus on the mode transitions. Rather, we focus here on how to reduce the energy consumption during the execution of a single operating mode. The chapter is organized as follows.

- In Section 4.5, we tackle the problem of choosing the smallest (or so) operating frequency common to every processor such that all the deadlines are met while running the application. The proposed techniques are offline and provide a static result in the sense that the computed speeds do not change over time. Such offline solutions are sufficient to significantly reduce the energy consumption; however, due to the discrepancy between Worst-Case Execution Time (WCET) and Actual-Case Execution Time (ACET) [31], they usually lead to pessimistic results.

- In Section 4.6.2, we present our online slack reclamation scheme named MORA. According to [26] and to the best of our knowledge, this is the first work which addresses a slack reclamation scheme targeting sporadic tasks and identical multiprocessor platforms. Although most previous studies on multiprocessor energy-efficient scheduling assumed that the actual execution time of a task is equal to its Worst-Case Execution Time (WCET), such that those in [1, 8, 24, 70] for instance, this work is motivated by the scheduling of tasks in practice, where tasks might usually complete earlier than their WCET [7, 77]. The proposed algorithm MORA
is an online scheme which exploits early task completions by using the “unused” time to reduce the execution speed of the next running jobs as much as possible. MORA has been inspired from the uniprocessor “Dynamic Reclaiming Algorithm” (DRA) proposed in [7], but the way in which it profits from the unused time is very different from the DRA since MORA takes into consideration the application-specific consumption profile of the tasks. Extending the concept of the algorithm DRA to the multiprocessor context was a real challenge, mainly because DVFS multiprocessor scheduling algorithms have to be extremely careful while modifying the processors speeds. Indeed, in a multiprocessor environment, modifying the original schedule can expose the resulting schedule to scheduling anomalies (see Chapter 1, page 37 for a brief introduction to scheduling anomalies).

In Section 4.6.3, we propose our online scheme MOTE that takes advantage of unused time slots to further reduce the energy consumption. This algorithm analyzes the produced schedule at run-time and anticipates the future intervals of time during which the processors will idle. Then, MOTE reduces these idle times by dynamically slowing down the speed of the processors. We proved that these speed reductions do not jeopardize the schedulability of the application and provide significant energy savings. This algorithm MOTE can be considered as an extension to the multiprocessor case of a previous proposal of Shin and Shoi in [62], which is usually referred to as “One Task Extension” (OTE). Once again, extending the algorithm OTE to multiprocessor platforms was a real challenge because of the presence of scheduling anomalies in multiprocessor environments.

In Section 4.6.4, we explain how the algorithms MORA and MOTE can be combined in order to increase the energy savings.

In Section 4.7, we report on some simulation results that analyse the average energy savings provided by our proposed solutions. These simulations are based on realistic data and take into account real characteristics of the processors.

Finally in Section 4.8, we introduce future research directions and we conclude.

4.5 Offline speed determination

The offline speed determination problem consists in determining the speed(s) of every processor at system design-time, i.e., before the execution of the application. For table-driven schedulers (see Chapter 1 on page 29 for a definition), since the entire schedule
is known beforehand, it is possible to determine multiple speeds for each processor, as well as the instants at which the processors have to change their execution speed during the execution of the application. However, such schedulers are not considered in this thesis and this particular speed determination problem is hence out of the scope of this chapter. Here, we address the problem of determining one speed for each processor at system design-time (i.e., we design offline algorithms), such that the processors never change their assigned speed at run-time. We distinguish between the following two types of speed assignments. Actually, we say in the remainder of this chapter that we distinguish between two types of speed: identical and individual speeds, according to the following definition.

**Definition 4.4 (identical and individual speeds)**

A speed is said to be identical if it has to be used simultaneously by all the DVFS-identical processors of the platform. Otherwise, each DVFS-identical processor can operate at a speed different from that of the other processors and their assigned speed is said to be individual.

Clearly, dependent DVFS-identical processors support only identical speeds while independent processors support both identical and individual speeds. In this section, we propose three different speed assignment mechanisms.

1. Our first algorithm is presented in the following Section 4.5.1. It is a generic approach in the sense that this algorithm is compatible with any scheduler, as long as a schedulability test based on the tasks and platform characteristics is available. This method determines an identical speed $s$ such that the given task set is schedulable upon the given platform if all the processors run at speed $s$.

2. Second, we propose in Section 4.5.2 an algorithm specific to the scheduler EDF. This algorithm improves the result obtained from the generic approach given previously. That is, for the particular case of EDF, this method provides an identical speed that is never larger than that returned by the generic approach introduced in Section 4.5.1.

3. Finally, in Section 4.5.3, we propose another generic approach that determines a set of individual processor speeds. Like our first generic approach, this method is compatible with any scheduler, as long as a schedulability test based on the tasks and platform characteristics is available.
4.5 Offline speed determination

4.5.1 Determination of an identical speed: a generic approach

In this section, we address the problem of determining an identical speed $s \leq s^{\text{max}}$ such that (i) $s$ is as low as possible and (ii) all the deadlines are met if every processor runs at speed $s$. The solution proposed here is very general in the sense that it can be adapted to any scheduler as long as a schedulability test based on the tasks and platform characteristics is available for that scheduler. This generic approach consists in three consecutive steps, each described below. From this point forward, the notation $S$ refers to the considered scheduling algorithm.

**Step 1.** The first step consists in choosing an existing schedulability test for $S$ on multiprocessor platforms. For instance, concerning the popular schedulers DM or EDF, many sufficient schedulability tests have been proposed in the literature (see for instance [9, 10, 11, 12, 14, 15, 16, 22] for EDF and [9, 17, 18] for DM). Our approach is presented while considering the scheduler EDF and the “simple” schedulability Test 4.1 (given below) proposed by Bertogna, Cirinei and Lipari in [22].

**Schedulability Test 4.1 (From Bertogna et al. [22])**

Using EDF, any set $\tau$ of sporadic constrained-deadline tasks satisfying

$$\delta_{\text{sum}}(\tau) \leq m - (m - 1) \cdot \delta_{\text{max}}(\tau) \quad (4.1)$$

is schedulable upon $m$ identical processors.

**Step 2.** As we can see, the speed of the processors is not explicitly specified in the above test. Rather, Inequality 4.1 implicitly assumes that the $m$ processors are running at maximal processor speed. This second step consists in adapting this test in such a manner that it explicitly specifies the identical speed (say $s$) of the processors. According to our interpretation of the speed, if all the processors run at speed $s$ then the WCET $C_i$ of every task $\tau_i$ becomes $\frac{C_i}{s}$, implying that all the terms $C_i$ have to be replaced with $\frac{C_i}{s}$ in the chosen schedulability test (here, Test 4.1). Notice that this modification can involve the modification of other notations. For instance in Test 4.1, $\delta_{\text{max}}(\tau)$ and $\delta_{\text{sum}}(\tau)$ have to be replaced with $\frac{\delta_{\text{max}}(\tau)}{s}$ and $\frac{\delta_{\text{sum}}(\tau)}{s}$, respectively. Then, this second step ends by “isolating” $s$. Since most tests proposed in the literature are sufficient, they are often expressed as one or several inequalities and isolating $s$ often boils down to rewrite these inequalities with $s$ alone in any side of the inequality(ies). This is the case for the simple
Test 4.1 that we consider here. Indeed, this second step of our approach yields the following Test 4.2.

**Schedulability Test 4.2 (From previous work [57])**

Using EDF, any set $\tau$ of sporadic constrained-deadline tasks is schedulable upon $m$ DVFS-identical processors running at speed $s$, provided

$$s \geq \delta_{\text{max}}(\tau) + \frac{\delta_{\text{sum}}(\tau) - \delta_{\text{max}}(\tau)}{m}$$

**Proof**

Note that from Expression 4.2, $s$ is never lower than $\delta_{\text{max}}(\tau)$, which is a necessarily condition to ensure the schedulability of the application whatever the considered scheduling algorithm. By assumption, the worst-case execution time of every task $\tau_i$ is $\frac{C_i}{s}$ at any speed $s$ such that $s_{\text{min}} \leq s \leq s_{\text{max}}$. Thus, the maximum and generalized density of the application is $\frac{\delta_{\text{max}}(\tau)}{s}$ and $\frac{\delta_{\text{sum}}(\tau)}{s}$, respectively. From Test 4.1, the following condition is therefore *sufficient* to ensure the schedulability of the application using EDF:

$$\frac{\delta_{\text{sum}}(\tau)}{s} \leq m - (m - 1) \cdot \frac{\delta_{\text{max}}(\tau)}{s}$$

And rewriting this yields

$$s \geq \delta_{\text{max}}(\tau) + \frac{\delta_{\text{sum}}(\tau) - \delta_{\text{max}}(\tau)}{m}$$

which states the proof.

**Step 3.** This resulting Test 4.2 allows us for the third and last step of our approach. Based on this test, the third step consists in establishing a mathematical expression that provides an identical speeds as low as possible, and such that $s$ guarantees the schedulability of the application. According to Test 4.2, the lowest identical processor speed that guarantees the schedulability while using EDF is given by $s_{\text{ident}}^{edf}$ where

$$s_{\text{ident}}^{edf} \overset{\text{def}}{=} \max \left\{ s_{\text{min}}, \delta_{\text{max}}(\tau) + \frac{\delta_{\text{sum}}(\tau) - \delta_{\text{max}}(\tau)}{m} \right\}$$

(4.3)
4.5 Offline speed determination

The approach was straightforward for this example, because the schedulability Test 4.1 that we considered initially is simple. However, some schedulability tests proposed in the literature are much more complex than Test 4.1. For instance, let us consider the following test proposed by Baker in [9] for the scheduler DM.

**Schedulability Test 4.3 (from Theodore Baker [9])**

Using DM, any set $\tau$ of sporadic constrained-deadline tasks is schedulable upon an identical multiprocessor platform composed of $m$ processors if, for every task $\tau_k$,

$$ \frac{\sum_{\tau_i \in \tau} \alpha_{i,k}(\tau)}{1 - \delta_k} \leq m $$

where

$$ \alpha_{i,k}(\tau) = \begin{cases} U_i \cdot \left(1 + \frac{T_i - C_i}{D_k}\right) & \text{if } \delta_{\text{max}}(\tau) \geq U_i \\ U_i \cdot \left(1 + \frac{T_i - C_i}{D_k}\right) + \frac{C_i - \delta_{\text{max}}(\tau) \cdot T_i}{D_k} & \text{otherwise} \end{cases} $$

As we can see, modifying the test so that the identical speed $s$ is explicitly specified (by replacing all the terms $C_i$ with $\frac{C_i}{s}$) is still straightforward, but isolating $s$ from the rest of the inequality is much more difficult since $s$ should appear in the discontinuous function $\alpha_{i,k}(\tau)$. Therefore, for schedulability tests of such complexity, the second step of our approach consists only in modifying the test so that $s$ is explicitly specified (without isolating $s$). With Test 4.3, this leads to the following sufficient schedulability Test 4.4.

**Schedulability Test 4.4**

Any set $\tau$ of sporadic constrained-deadline tasks is schedulable using DM upon $m$ DVFS-identical processors running at speed $s$, provided that $\forall k$, $1 \leq k \leq n$,

$$ \frac{\sum_{\tau_i \in \tau} \alpha_{i,k}(\tau, s)}{1 - \frac{s}{s}} \leq m $$

where

$$ \alpha_{i,k}(\tau, s) = \begin{cases} \frac{U_i}{s} \cdot \left(1 + \frac{sT_i - C_i}{sD_k}\right) & \text{if } \delta_{\text{max}}(\tau) \geq U_i \\ \frac{U_i}{s} \cdot \left(1 + \frac{sT_i - C_i}{sD_k}\right) + \frac{C_i - \delta_{\text{max}}(\tau) \cdot T_i}{sD_k} & \text{otherwise} \end{cases} $$

As mentioned above, isolating $s$ from the rest of the inequality is not straightforward.
in this case, and the third step of our approach is therefore different than that presented earlier. Instead of deriving a tailor-made expression as provided by Expression 4.3, we introduce the following function $\text{sched}(S, \tau, m, s)$ which is based on the test resulting from the previous step (here, Test 4.4):

$$
\begin{align*}
\text{sched}(S, \tau, m, s) & \overset{\text{def}}{=} \\
& \begin{cases} \\
\text{True} & \text{if } \tau \text{ is schedulable by } S \text{ upon } m \text{ identical processors running at speed } s \\
\text{False} & \text{otherwise}
\end{cases}
\end{align*}
$$

**Algorithm 5: Determination of an identical speed.**

**Input:**
- A set $\tau$ of real-time tasks,
- A number $m$ of DVFS-identical processors,
- A predictable scheduler $S$,
- A function $\text{sched}(S, \tau, m, s)$ based on any schedulability test for $S$,
- A maximal processor speed $s_{\text{max}}$,
- A minimal processor speed $s_{\text{min}}$.

**Output:** An identical processors speed

```plaintext
begin
1    if (sched(S, \tau, m, s_{\text{max}}) == False) then return \# APP_NOT_SCHEDULABLE;
2    s_{\text{current}} \leftarrow s_{\text{max}} ;
3    \textbf{while } (s_{\text{current}} \geq s_{\text{min}} \text{ and } \text{sched}(S, \tau, m, s_{\text{current}})) \textbf{ do}
4        s_{\text{current}} \leftarrow s_{\text{current}} - s_{\text{step}} ;
5    \textbf{end while}
6    s_{\text{current}} \leftarrow s_{\text{current}} + s_{\text{step}} ;
7    \textbf{return } s_{\text{current}} ;
8 end
```

Finally, we implement this function in the generic method given by Algorithm 5, where $s_{\text{current}}$ denotes the identical speed to be returned. At line 3, $s_{\text{current}}$ is initially set to $s_{\text{max}}$. Then from line 4 to 6, $s_{\text{current}}$ is decremented as long as (i) $s_{\text{current}}$ is not lower than $s_{\text{min}}$ and (ii) the schedulability of the application is guaranteed by the function $\text{sched}(S, \tau, m, s_{\text{current}})$. The decrement $s_{\text{step}}$ can be set to any arbitrarily small value—the smaller $s_{\text{step}}$, the longer the computation time of the algorithm, but more accurate are the results\(^1\). At line 7, $s_{\text{current}}$ is incremented by $s_{\text{step}}$ since the loop-condition at line 4 was not satisfied. Finally, line 8 returns the computed speed $s_{\text{current}}$. Figure 4.2 illustrates

\(^1\)We have seen that practical processors provide only a finite set of speeds. For such practical processors,
and summarizes the entire approach that we introduced in this section.

### 4.5.1.1 Improvements of this generic approach

We propose here two solutions that improve the generic method presented above.

**First improvement.** It concerns the computation time of Algorithm 5. Indeed, the computation time can be improved by using a dichotomic search instead of iteratively decrementing the speed \( s_{\text{current}} \) by the constant rate \( s_{\text{step}} \). However, this optimization makes sense only in *theory* because from a practical point of view, the number of speeds available to the processors is limited. In the Appendix (on page 347), we present the characteristics of 5 commercial processors and the number of speeds that they provide lies between 4 (see the PowerPC-405LP) and 11 (see the StrongARM SA-1100). The we assume that at each execution of the while-loop, \( s_{\text{step}} \) is set to the difference between \( s_{\text{current}} \) and the lower available speed that directly follows.
sequential approach given by Algorithm 5 is therefore appropriate for such a limited number of speeds.

**Second improvement.** It concerns the schedulability test chosen in the first step of our approach. Indeed, more accurate the test, more efficient the approach. Consequently, the approach can be improved by using a combination of schedulability tests rather than a single test. For instance, concerning EDF, we gathered several schedulability tests in Chapter 3 on page 215 and we introduced the following resulting Test 4.5.

**Schedulability Test 4.5**

A set of sporadic constrained-deadline tasks $\tau$ is schedulable using global-EDF upon an identical multiprocessor platform composed of $m$ processors if one of the following is true:

- **[BCL] condition** (from Bertogna, Cirinei and Lipari in [22]):

  \[ \delta_{\text{sum}}(\tau) \leq m - (m - 1) \cdot \delta_{\text{max}}(\tau) \]

- **[BAK] condition** (from Baker in [9, 10]): for every task $\tau_k \in \tau$, there exists $\lambda \in \{\delta_k\} \cup \{U_i \mid U_i \geq \delta_k, \tau_i \in \tau\}$, such that

  \[ \sum_{\tau_i \in \tau} \min\{\beta^k_{\tau}(i), 1\} \leq m \cdot (1 - \lambda) + \lambda \]

  where

  \[ \beta^k_{\tau}(i) \overset{\text{def}}{=} \begin{cases} U_i \cdot \left(1 + \max\{0, \frac{T_i - D_i}{D_k}\}\right) & \text{if } U_i \leq \lambda \\ U_i \cdot \left(1 + \frac{T_i}{D_k}\right) - \lambda \cdot \frac{D_i}{D_k} & \text{otherwise} \end{cases} \]

- **[BCL2] condition** (from Bertogna, Cirinei and Lipari in [22]): for each task $\tau_k$, one of the two following is true:

  \[ \sum_{i \neq k} \min\{\beta_k(i), 1 - \delta_k\} < m \cdot (1 - \delta_k) \]

  or

  \[ \sum_{i \neq k} \min\{\beta_k(i), 1 - \delta_k\} = m \cdot (1 - \delta_k) \text{ and } \exists i \neq k : 0 < \beta_k(i) \leq 1 - \delta_k \]

  where
4.5 Offline speed determination

\[
\beta_k(i) \overset{\text{def}}{=} \frac{\text{dbf}(\tau_i, C_k) + \min\left\{ C_i, \max\left\{ 0, D_k - \text{dbf}(\tau_i, C_k) \cdot \frac{T_i}{C_i} \right\} \right\}}{D_k}
\]

and \( \text{dbf}(\tau_i, t) \) is defined as in Chapter 1, page 17.


\[
\text{LOAD}(\tau) \leq \frac{m^2}{2m-1} - (m - 1) \cdot \frac{\delta_{\max}(\tau)}{2}
\]

where \( \text{LOAD}(\tau) \) is defined as in Chapter 1, page 17.

Following the second step of our approach, we can rewrite this schedulability test by replacing every term \( C_i \) with \( \frac{C_i}{\tau} \), so that it takes into consideration any identical speed \( s \). This provides the following Test 4.6.

**Schedulability Test 4.6**

A set \( \tau \) of sporadic constrained-deadline tasks is schedulable by global-EDF upon \( m \) identical processors running at speed \( s \) if one of the following is true:

- From [BCL] condition:

\[
\frac{\delta_{\sum}(\tau)}{s} \leq m - (m - 1) \cdot \frac{\delta_{\max}(\tau)}{s}
\]

- From [BAK] condition: For every task \( \tau_k \in \tau \), there exists \( \lambda \in \left\{ \frac{\delta_k}{\tau} \right\} \cup \left\{ \frac{U_i}{\tau} \mid \frac{U_i}{\tau} \geq \frac{\delta_i}{\tau}, \tau_i \in \tau \right\} \), such that:

\[
\sum_{\tau_i \in \tau} \min \left\{ \alpha_{k}^\lambda(i, s), 1 \right\} \leq m \cdot (1 - \lambda) + \lambda
\]

where

\[
\alpha_{k}^\lambda(i, s) \overset{\text{def}}{=} \begin{cases} 
\frac{U_i}{\tau} \cdot (1 + \max\left\{ 0, \frac{T_i - D_i}{D_k} \right\}) & \text{if } \frac{U_i}{\tau} \leq \lambda \\
\frac{U_i}{\tau} \cdot (1 + \frac{T_i}{D_k}) - \lambda \cdot \frac{D_i}{D_k} & \text{otherwise}
\end{cases}
\]
From [BCL2] condition: \( \forall \tau_k \in \tau \) one of the two following is true:

\[
\sum_{i:k} \min \left\{ \beta_k(i,s), 1 - \frac{\delta_k}{s} \right\} < m \cdot \left( 1 - \frac{\delta_k}{s} \right)
\]

or

\[
\sum_{i:\neq k} \min \left\{ \beta_k(i,s), 1 - \frac{\delta_k}{s} \right\} = m \cdot \left( 1 - \frac{\delta_k}{s} \right) \quad \text{and} \quad \exists i \neq k : 0 < \beta_k(i,s) \leq 1 - \frac{\delta_k}{s}
\]

where

\[
\beta_k(i,s) \overset{\text{def}}{=} \frac{\text{DBF}(\tau_i, s, \frac{C_i}{s}) + \min \left\{ C_i, \max \left\{ 0, D_k - \text{DBF}(\tau_i, s, \frac{C_i}{s}) \cdot \frac{T_i s}{C_i} \right\} \right\}}{D_k}
\]

and \( \text{DBF}(\tau_i, s, t) \) is defined as in Expression 4.4.

From [BB] condition:

\[
\text{LOAD}(\tau, s) \leq \frac{m^2 - (m - 1) \cdot \frac{\delta_{\text{max}}(t)}{s}}{2}
\]

where

\[
\text{LOAD}(\tau, s) \overset{\text{def}}{=} \max_{t>0} \left( \frac{\sum_{\tau_i \in \tau} \text{DBF}(\tau_i, s, t)}{t} \right)
\]

and \( \text{DBF}(\tau_i, s, t) \) is defined as

\[
\text{max} \left\{ 0, \left( \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1 \right) \cdot \frac{C_i}{s} \right\} \quad (4.4)
\]

Following the third step of our approach, this resulting Test 4.6 can be rewritten as a function \( \text{sched}(S, \tau, m, s) \) and can be incorporated into Algorithm 5.

### 4.5.2 Determination of an identical speed: a method specific to EDF

In this section, we propose an identical speed determination algorithm specific to the scheduler EDF. Actually, this section considers an adaptation of EDF, named \( \text{EDF}^{(k)} \), for which it was proven in [33] that any set of tasks schedulable by \( \text{EDF}^{(k)} \) is also schedulable by EDF (requiring however a small modification in the tasks characteristics). The algorithm \( \text{EDF}^{(k)} \) was first introduced in [33] with the intention to reduce the number of processors needed to successfully schedule a given set of tasks in regard to a sufficient test. Below, we propose another schedulability analysis of \( \text{EDF}^{(k)} \), in which the number
of processors is fixed beforehand. The aim of our analysis is not to allow for using a lower amount of processors, but a lower identical processor speed. Basically, EDF\(^{(k)}\) works as follows: assuming that the tasks are ordered by non-increasing task densities, i.e., \(\delta_i \geq \delta_{i+1} \forall i\), EDF\(^{(k)}\) (with \(1 \leq k \leq m\)) assigns priorities to jobs according to the following rule:

**For all** \(i < k\): all the jobs issued from the task \(\tau_i\) receive the highest priority (ties are broken arbitrarily)—This is trivially achieved within an EDF implementation by setting the deadline of each such \(\tau_i\) to \(-\infty\).

**For all** \(i \geq k\): the priority of every job issued from \(\tau_i\) is determined according to EDF (ties are again broken arbitrarily).

That is, Algorithm EDF\(^{(k)}\) assigns the highest priority to the jobs generated by the \((k - 1)\) tasks of highest densities, and assigns priorities according to EDF to the jobs generated by all the other tasks (thus, “pure” EDF is EDF\(^{(1)}\)). In the following we show that, with an appropriate value of \(k\), using EDF\(^{(k)}\) instead of EDF allows the processors to use an identical speed that can be lower (or at least never larger) than that returned by our generic approach described in the previous section. But first, let us introduce the notation \(\tau^{(i)}\) that refers to the subset of tasks composed of the \((n - i + 1)\) tasks of smallest densities in \(\tau\), i.e., \(\tau^{(i)} \overset{\text{def}}{=} \{\tau_{i},\tau_{i+1},\ldots,\tau_{n}\}\). According to this notation, it holds that \(\tau \equiv \tau^{(1)}\). The following Lemma 4.1 establishes a lower-bound on the identical processor speed for EDF\(^{(k)}\), where \(k\) is fixed beforehand. Then we prove in Corollary 4.2 that this lower-bound is never larger than the speed returned by our generic approach.

**Lemma 4.1 (derived from Theorem 2 in [57])**

Using EDF\(^{(k)}\) (for \(1 \leq k \leq m\)), any set \(\tau\) of sporadic constrained-deadline tasks is schedulable upon \(m\) DVFS-identical processors running at speed \(s_{\text{ident}}\) provided

\[
s_{\text{ident}} \geq \max \left\{ \delta_1, s_{\text{ident}}(\tau^{(k)}, m - k + 1) \right\}
\]
Once the tasks are ordered by non-increasing densities, $\text{EDF}^{(k)}$ splits the set of tasks into two subsets: the privileged tasks $\tau_1, \ldots, \tau_{k-1}$ and the unprivileged tasks $\tau_k, \tau_{k+1}, \ldots, \tau_n$. Here, $k = 3$.

Each privileged task is assigned to a dedicated processor.

The unprivileged tasks are scheduled by $\text{EDF}$ upon the remaining non-dedicated processors.

**Figure 4.3:** The main idea behind $\text{EDF}^{(k)}$. 
4.5 Offline speed determination

where \(s^\text{def}_{\text{ident}}(\tau^{(k)}, m - k + 1)\) denotes any identical speed that can schedule the task set \(\tau^{(k)}\) by EDF upon \((m - k + 1)\) processors without missing any deadline.

Proof

Let \(\overline{\text{EDF}}^{(k)}\) be the same scheduling algorithm as \(\text{EDF}^{(k)}\) except that \(\overline{\text{EDF}}^{(k)}\) dedicates an individual processor to the \((k - 1)\) tasks of highest densities and schedules the \((n - k + 1)\) remaining tasks by EDF upon the remaining \((m - k + 1)\) processors. Hereafter, the \((k - 1)\) tasks with the highest densities are called the privileged tasks (since each one benefits from an individual processor) and their assigned processors are called the dedicated processors. Symmetrically, the \((n - k + 1)\) remaining tasks are called unprivileged tasks and the \((m - k + 1)\) remaining processors are called non-dedicated processors. The proof is divided into two parts. First, we show that \(\tau\) is schedulable by \(\overline{\text{EDF}}^{(k)}\) upon the \(m\) processors at speed \(s^\text{def}_{\text{ident}}\) provided \(s^\text{def}_{\text{ident}} \geq \max\{\delta_1, s^\text{def}_{\text{ident}}(\tau^{(k)}, m - k + 1)\}\). Second, we show that any application \(\tau\) schedulable by \(\overline{\text{EDF}}^{(k)}\) is also schedulable by \(\text{EDF}^{(k)}\).

Part 1. By definition, \(\overline{\text{EDF}}^{(k)}\) splits the set of tasks into two exhaustive and mutually exclusive subsets: the subset of privileged tasks \(\{\tau_1, \tau_2, \ldots, \tau_{k-1}\}\) and the subset of unprivileged tasks \(\tau^{(k)}\) (see Figure 4.3a, where 8 tasks are allocated to 4 processors by \(\overline{\text{EDF}}^{(3)}\)). Then, each privileged task is assigned to a dedicated processor (see Figure 4.3b) and, for obvious reason, the minimum admissible speed of each dedicated CPU is the density of its assigned privileged task. As a result, the minimum common speed \(s\) of all the dedicated processors is the maximal density of the privileged tasks, i.e., \(s \geq \delta_{\text{max}}(\{\tau_1, \tau_2, \ldots, \tau_{k-1}\})\), and since the tasks are indexed by non-increasing order of task densities, this condition can be rewritten as

\[
s \geq \delta_1 \tag{4.5}
\]

On the other hand, the unprivileged tasks are scheduled upon the \((m - k + 1)\) non-dedicated processors—the processors that do not execute a privileged task (see Figure 4.3c)—and we know by definition of \(s^\text{def}_{\text{ident}}(\tau^{(k)}, m - k + 1)\) that the subset \(\tau^{(k)}\) of tasks is schedulable by EDF upon \((m - k + 1)\) processors running at speed \(s\) provided

\[
s \geq s^\text{def}_{\text{ident}}(\tau^{(k)}, m - k + 1) \tag{4.6}
\]

From Inequalities 4.5 and 4.6, it results that the whole application \(\tau\) is schedulable by \(\overline{\text{EDF}}^{(k)}\) upon the \(m\) processors running at speed \(s^\text{def}_{\text{ident}}\) provided

\[
s^\text{def}_{\text{ident}} \geq \max\{\delta_1, s^\text{def}_{\text{ident}}(\tau^{(k)}, m - k + 1)\}
\]
CHAPTER 4. EXPLOITING THE DVFS FRAMEWORK

Part 2. In this second part, we show that any application $\tau$ schedulable by $\text{EDF}^{(k)}$ is also schedulable by $\text{EDF}^{(k)}$.

For the privileged tasks. $\text{EDF}^{(k)}$ assigns the highest priority to the $(k - 1)$ privileged tasks. Since $k \leq m$, it holds that $(k - 1) < m$ and thus, all the jobs issued from these $(k - 1)$ privileged tasks are dispatched to a processor as soon as they are released. Consequently, the instants at which these jobs complete are exactly the same in the schedules produced by $\text{EDF}^{(k)}$ and $\overline{\text{EDF}}^{(k)}$. Since $\overline{\text{EDF}}^{(k)}$ meets the deadlines of all these jobs, it holds that $\text{EDF}^{(k)}$ meets the deadlines of all these jobs as well.

For the unprivileged tasks. In the schedule generated by $\text{EDF}^{(k)}$, the unprivileged tasks are only executed upon the $(m - k + 1)$ non-dedicated processors and every interval of time during which at least one dedicated processor idles can therefore be considered as lost.

However, by using $\text{EDF}^{(k)}$, the unprivileged tasks may be executed upon any of the $m$ processors and therefore, they can profit from all these intervals of time “lost” on the $(k - 1)$ dedicated processors. Thereby, the cumulated processing times executed on the non-dedicated processors is lower while using $\text{EDF}^{(k)}$ instead of $\overline{\text{EDF}}^{(k)}$ and the proof follows from the predictability of any global, preemptive, work-conserving and FJP scheduler (including $\text{EDF}^{(k)}$) showed in [36, 37, 38].

From both parts of this proof, we can conclude that the application $\tau$ can be successfully scheduled by $\text{EDF}^{(k)}$ upon $m$ processors running at speed $s^{\text{id}}_{\text{ident}}$ provided

$$s^{\text{id}}_{\text{ident}} \geq \max \left\{ \delta_1, s^{\text{id}}_{\text{ident}}(\tau^{(k)}, m - k + 1) \right\}$$

Corollary 4.1 (derived from Corollary 2 in [57])

Let

$$\ell = \arg\min_{k \in [1, \ldots, m]} \left\{ s^{\text{id}}_{\text{ident}}(\tau^{(k)}, m - k + 1) \right\}$$
Using EDF(\(\ell\)), any set \(\tau\) of sporadic constrained-deadline tasks is schedulable upon \(m\) DVFS-identical processors running (at least) at speed \(s_{\text{ident}}^{\text{edf}}(\ell)\) where

\[
s_{\text{ident}}^{\text{edf}}(\ell) \overset{\text{def}}{=} \max \left\{ \delta_1, s_{\text{ident}}^{\text{edf}}(\tau^{(\ell)}, m - \ell + 1) \right\}
\]  

(4.7)

**Proof**

The proof is a direct consequence of Lemma 4.1.

**Corollary 4.2**

The identical speed \(s_{\text{ident}}^{\text{edf}}(\ell)\) defined by Expression 4.7 is never larger than the speed \(s_{\text{ident}}^{\text{edf}}(\tau, m)\) obtained from our generic approach described in the previous section, i.e.,

\[
s_{\text{ident}}^{\text{edf}}(\ell) \leq s_{\text{ident}}^{\text{edf}}(\tau, m)
\]

**Proof**

The proof directly follows from the definition of \(s_{\text{ident}}^{\text{edf}}(\ell)\) in Expression 4.7. Indeed, \(\ell \in [1, m]\) is the value of \(k\) that minimizes \(s_{\text{ident}}^{\text{edf}}(\tau^{(k)}, m - k + 1)\). Therefore, it holds that

\[
s_{\text{ident}}^{\text{edf}}(\ell) \leq \max \left\{ \delta_1, s_{\text{ident}}^{\text{edf}}(\tau^{(1)}, m - 1 + 1) \right\}
\]

and since we know that no application is schedulable at a speed lower than \(\delta_1\), it holds that

\[
s_{\text{ident}}^{\text{edf}}(\tau^{(1)}, m) \geq \delta_1
\]

Finally, since \(\tau^{(1)} \overset{\text{def}}{=} \tau\) by definition of \(\tau^{(\ell)}\), we have

\[
s_{\text{ident}}^{\text{edf}}(\ell) \leq s_{\text{ident}}^{\text{edf}}(\tau, m)
\]

Algorithm 6 gives a pseudo-code of our identical speed determination algorithm for EDF(\(k\)). Recall that this algorithm is used at system design-time. The returned speed is then assigned to every processor and each processor maintains this assigned speed constant at run-time. Before invoking this algorithm, we assume that the number of
Algorithm 6: Determination of an identical speed for EDF<sup>(k)</sup>

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>begin</td>
</tr>
<tr>
<td>2</td>
<td>( k_{\text{opt}} \leftarrow 1; )</td>
</tr>
<tr>
<td>3</td>
<td>( s_{\text{ident}} \leftarrow s_{\text{max}}; )</td>
</tr>
<tr>
<td>4</td>
<td>( s_{\text{limit}} \leftarrow \max{s_{\text{min}}, \delta_1}; )</td>
</tr>
<tr>
<td>5</td>
<td>for ( (k := 1; k \leq m, s_{\text{ident}} &gt; s_{\text{limit}}; + + k) ) do</td>
</tr>
<tr>
<td>6</td>
<td>( s \leftarrow \max{\delta_1, s_{\text{edf}}^{\text{ident}}(\tau(k), m - k + 1)}; )</td>
</tr>
<tr>
<td>7</td>
<td>if ( (s &lt; s_{\text{ident}}) ) then</td>
</tr>
<tr>
<td>8</td>
<td>( s_{\text{ident}} \leftarrow s; )</td>
</tr>
<tr>
<td>9</td>
<td>( k_{\text{opt}} \leftarrow k; )</td>
</tr>
<tr>
<td>10</td>
<td>if ( (s_{\text{ident}} &lt; s_{\text{limit}}) ) then ( s_{\text{ident}} \leftarrow s_{\text{limit}}; )</td>
</tr>
<tr>
<td>11</td>
<td>end if</td>
</tr>
<tr>
<td>12</td>
<td>end for</td>
</tr>
<tr>
<td>13</td>
<td>foreach ( \tau_i \in {\tau_1, \ldots, \tau_{k_{\text{opt}} - 1}} ) do ( D_i \leftarrow -\infty; )</td>
</tr>
<tr>
<td>14</td>
<td>return ( s_{\text{ident}}; )</td>
</tr>
<tr>
<td>15</td>
<td>end</td>
</tr>
</tbody>
</table>

processors is sufficient to schedule the application \( \tau \) at the maximal processor speed \( s_{\text{max}} \). This is the reason why the returned speed \( s_{\text{ident}} \) is initially set to \( s_{\text{max}} \) (line 3). Then, the algorithm searches the minimal speed by sweeping the value of \( k \) between 1 and \( m \) (line 5 to line 12). The function \( s_{\text{edf}}^{\text{ident}}(\tau, m) \) invoked at line 6 can be any function returning an identical processor speed \( s \) such that the task set \( \tau \) is schedulable by EDF upon \( m \) of processors running at speed \( s \). For instance, this function could be the generic approach introduced in the previous section. Finally at line 13, assuming that \( \tau \) will be ultimately scheduled by EDF, the algorithm sets the deadline of the \( (k - 1) \) highest-density tasks to \(-\infty\) so that EDF will assign the highest priority to these tasks.

### 4.5.3 Determination of individual processor speeds

The problem of determining individual processor speeds can be reduced to the scheduling problem upon uniform platforms. Indeed, once the speed of every processor is determined, each processor of the platform runs at its assigned speed and keeps it con-
4.5 Offline speed determination

stant at run-time. Consequently, the platform can be seen as composed of processors with distinct computing capability (i.e., an uniform platform). The approach that we propose here is very similar to that introduced in Section 4.5.1. Also, this method is generic and works for any predictable scheduler $S$, as long as a schedulability test for uniform platforms is available for $S$. Actually, the approach proposed here differs from that of Section 4.5.1 only in the generic algorithm used at the very end. Indeed, this approach is also divided into three steps, where the first step consists in choosing an appropriate schedulability test. For instance, assuming the scheduler DM, one can use the following test.

**Schedulability Test 4.7 (from Sanjoy Baruah and Joël Goossens [18])**

Using DM, any set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of $n$ sporadic constrained-deadline tasks (assuming $D_i \leq D_{i+1}$ $\forall i$) is schedulable upon any $m$-processors uniform platform $\pi = \{s_1, s_2, \ldots, s_m\}$ (assuming $s_i \geq s_{i+1}$ $\forall i$) provided that for all $k$, $1 \leq k \leq n$,

$$2 \cdot \text{LOAD}(k) + v_k \cdot \max_{i=1}^k \delta_i \leq \mu_k$$

where

$$\text{LOAD}(k) \overset{\text{def}}{=} \max_{t \geq 0} \left\{ \frac{\sum_{i=1}^k \text{DBF}(\tau_i, t)}{t} \right\}$$

$$\text{DBF}(\tau_i, t) \overset{\text{def}}{=} \max \left\{ 0, \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1 \right\} \cdot C_i$$

$$\mu_k \overset{\text{def}}{=} \sum_{i=1}^m s_i - m \max_{i=1}^m \left\{ \frac{\sum_{j=i+1}^m s_j}{s_i} \right\} \cdot \max_{i=1}^\ell \delta_i$$

$$v_k \overset{\text{def}}{=} \max\{\ell \mid 1 \leq \ell \leq n \text{ and } \sum_{j=1}^\ell s_j \leq \mu_k\}$$

Then, the second step consists in writing a binary function $\text{sched}(\pi = \{s_1, s_2, \ldots, s_m\}, S, \tau)$ based on the test chosen in the first step. This function is defined by:

$$\text{sched}(\pi, S, \tau) \overset{\text{def}}{=} \begin{cases} 
\text{True} & \text{if } \tau \text{ is schedulable by } S \text{ upon the uniform platform } \pi \\
\text{False} & \text{otherwise}
\end{cases}$$
CHAPTER 4. EXPLOITING THE DVFS FRAMEWORK

Algorithm 7: Determination of individual speeds.

- **Input:**
  - A set $\tau$ of real-time tasks,
  - A platform $\pi$ composed of $m$ DVFS-identical processors,
  - A scheduler $S$,
  - A function $\text{sched}(S, \tau, m)$ based on any schedulability test for $S$,
  - An initial identical processor speed $s_{\text{ident}}$.

- **Output:** A set $\{s_1, s_2, \ldots, s_m\}$ of individual processor speeds

```
begin
foreach $\pi_k \in \pi$ do
  $s_k \leftarrow s_{\text{ident}}$;
  Unmark $\pi_k$;
end foreach
while ($\exists \pi_i \in \pi$ such that $\pi_i$ is not marked) do
  $\pi_{\text{unmark}} \leftarrow \phi$;
  foreach $\pi_i \in \pi$ such that $\pi_i$ is not marked do
    $\pi_{\text{unmark}} \leftarrow \pi_{\text{unmark}} \cup \{\pi_i\}$;
    $\ell \leftarrow \underset{k|\pi_k \in \pi_{\text{unmark}}}{\text{argmax}} \{s_k\}$;
    $s_{\ell} \leftarrow s_{\ell} - s_{\text{step}}$;
    if ($\text{sched}(\pi = \{s_1, s_2, \ldots, s_m\}, S, \tau) == \text{False}$) then
      $s_{\ell} \leftarrow s_{\ell} + s_{\text{step}}$;
      Mark $\pi_{\ell}$;
    else
      Unmark all the processors;
    end if
  end foreach
end while
return $\{s_1, s_2, \ldots, s_m\}$;
end
```

Finally, the third step consists in incorporating this function into a generic algorithm that iterates until finding an appropriate speeds assignment. A pseudo-code of this generic algorithm is given by Algorithm 7, which is inspired from the “Waterfall algorithm” proposed in [46] in the uniprocessor context. Initially, the speed $s_k$ of every processor $\pi_k$ is set to an identical speed $s_{\text{ident}}$ (line 3) passed as an argument to the algorithm. We assume that this identical speed $s_{\text{ident}}$ has been determined beforehand using an approach such as the generic one proposed in section 4.5.1 for instance. At line 4, all the processors are “unmarked” with the interpretation that a marked (resp. unmarked) processor can (resp. cannot) further reduce its speed. Then, while there is at least one unmarked processor (line 6), the algorithm performs the following treatment iteratively.
4.6 Online slack reclaiming algorithms

(lines 7 and 8). The set $\pi_{\text{unmark}}$ is determined. This set contains every unmarked processor.

(line 9). Select the unmarked processor $\pi_\ell \in \pi_{\text{unmark}}$ of maximum assigned speed:

$$\ell \leftarrow \arg\max_{k|\pi_k \in \pi_{\text{unmark}}} \{s_k\}$$

(line 10). Decrement the speed $s_\ell$ of the selected processor $\pi_\ell$:

$$s_\ell \leftarrow s_\ell - s_{\text{step}}$$

where $s_{\text{step}}$ is a variable set up by the system designer. Assuming that the processor speeds can be set to any non-integer value within $[s_{\text{min}}, s_{\text{max}}]$, the speed returned by this algorithm will be as accurate as $s_{\text{step}}$ is small (at the cost of a higher computing time). However, we have seen that practical processors have only a finite set of available speeds. For such practical processors, we assume that $s_{\text{step}}$ always corresponds to the gap between $s_\ell$ and the lower available speed that directly follows.

(line 11). Check the schedulability of the application with the current speeds assignment. If the application is not schedulable then:

(line 12). Undo the previous modification of $s_\ell$.

(line 13). Mark processor $\pi_\ell$. This indicates that the speed of $\pi_\ell$ cannot be further reduced with the current speeds assignment. Notice that initially, all the processors are unmarked.

(line 15). Otherwise, if the application is schedulable then unmark all the processors.

(line 18). If all the processors are marked then return the current speeds assignment.

4.6 Online slack reclaiming algorithms

4.6.1 Notion of slack time

The main drawback of offline energy-aware algorithms resides in the fact that they must consider that every job executes for its WCET, because no prior information is available about the actual execution times of these jobs. This assumption is therefore
unavoidable and severely restricts the energy savings that offline techniques can provide, unfortunately. In order to tackle this problem, authors have investigated how the energy consumption can be improved without any additional information about the actual execution times of the jobs and online algorithms have emerged.

As introduced previously, online algorithms (on the contrary to offline algorithms) are invoked at run-time. In the scope of the DVFS framework, online energy-aware algorithms manage the working voltage and frequency (i.e., the speed) of the processor(s) during the execution of the application. More precisely, they perform “local” adjustments of the processor speed at run-time so that the speed of each processor matches its current activity as closely as possible. The main idea can be summarized as follows: these online methods anticipate the interval of times during which a processor will idle and shrink this future “slack time” by slowing down the current speed of the concerned processor. That is the reason why these algorithms are named “slack reclaiming algorithms” in the literature (see [7] for instance).

The algorithms that we propose in this section are based on the concept of worst-case and actual schedules. The worst-case schedule is the schedule in which every job executes for its WCET whereas the actual schedule is the schedule which is actually produced at run-time. In this latter schedule, every job executes for its actual execution time (noted ACET for Actual-Case Execution Time) that can be lower than its WCET. To illustrate this difference between WCET and ACET, we refer to the work of V. Berten et al. [20] in which simulations were performed by using several workloads coming from video decoding using H.264. These simulations were carried out on a TI DaVinci DM6446 DVFS processor (please consult the paper [20] for more details). Figure 4.4 shows the distribution of the frame decoding times of the 4 video clips that they used, each with several thousands of frames. As we can observe in each of these figures, the ACET of a task can be much lower than its WCET.

The notions of worst-case and actual schedules are illustrated on Figure 4.5 through a simple example, in which 4 jobs $J_1, J_2, J_3, J_4$ are scheduled upon two identical processors using the job priority assignment $J_1 > J_2 > J_3 > J_4$. The WCET and the deadline of every job is 8 and 48 (respectively) and their actual execution times are 4, 5, 2 and 2,
Figure 4.4: Distribution of the number of cycles to decode different kinds of video, ranging from news streaming to complex 3D animations. X-axis: number of cycles, y-axis: probability.

respectively. The worst-case schedule of these jobs is depicted in Figure 4.5a whereas the actual one is depicted in Figure 4.5b. Both schedules are generated assuming that all the processors are running at speed $s^{\text{max}}$. Any interval of time during which a processor idles can be considered as “slack time” and in this study, we distinguish between two types of slack time: the internal and external slack time. Formal definitions of these concepts are given below.
CHAPTER 4. EXPLOITING THE DVFS FRAMEWORK

(a) Worst-case schedule at speed $s^{\text{max}} = 1$ using the job priority assignment $J_1 > J_2 > J_3 > J_4$.

(b) Actual schedule at speed $s^{\text{max}} = 1$ using the job priority assignment $J_1 > J_2 > J_3 > J_4$.

Figure 4.5: Illustration of the worst-case and actual schedules. The internal slack time is depicted in red and the external slack time is depicted in green.

Definition 4.5 (Internal slack time)

An interval of time $[t_1, t_2]$ is considered as “internal slack time” on processor $\pi_k$ if and only if

1. $\pi_k$ idles within $[t_1, t_2]$ in the actual schedule and,
2. $\pi_k$ does not idle within $[t_1, t_2]$ in the worst-case schedule.

Definition 4.6 (External slack time)

An interval of time $[t_1, t_2]$ is considered as “external slack time” on processor $\pi_k$ if and only if

1. $\pi_k$ idles within $[t_1, t_2]$ in the actual schedule and,
2. $\pi_k$ also idles within $[t_1, t_2]$ in the worst-case schedule.

Figure 4.5b illustrates these notions of internal and external slack time, where we highlighted the internal slack time in red and the external slack time in green. Since the
4.6 Online slack reclaiming algorithms

difference between the WCET and the ACET of the jobs can be important, the worst-case schedule can be very different from the actual one. The following two sections propose two online algorithms that aim to improve the energy savings by reclaiming this slack time. The first algorithm (named MORA) reclaims the internal slack time as much as possible and reduces the speed of the processors accordingly. The second one (named MOTE) reduces the processors speed by anticipating the external slack times.

4.6.2 The Multiprocessor Online Reclaiming Algorithm (MORA)

The algorithm MORA is an extension to multiprocessor platforms of the uniprocessor Dynamic Reclaiming Algorithm (DRA) proposed in [7]. As it will be showed through the simulation results of Section 4.7, offline speed determination algorithms can provide significant energy savings. Nevertheless, due to the important difference between the WCET and the ACET of the jobs, the worst-case schedule can be very different from the actual one. This difference clearly appears in Figure 4.5. Notice that these names and acronyms (WCET, ACET, worst-case schedule and actual schedule) will be widely used throughout this section.

Figure 4.6 depicts the same schedules as in Figure 4.5, but we suppose that an offline identical speed determination algorithm has set the speed to $\frac{1}{2}$. We can observe in this figures that the amount of internal slack times (depicted in red) is as large as the identical speed is low, whereas the amount of external slack times (depicted in green) decreases along with the identical speed. Based on this observation, our objective while thinking and designing MORA was to narrow the gap between the worst-case and actual schedules, by reclaiming the internal slack time as much as possible. Basically, MORA profits from the possible (and even likely) early completion of the jobs: whenever a job completes without having consumed its WCET: MORA reclaims the internal slack time left by this job and executes one of the next waiting jobs at a lower speed. More details about how MORA reduces the execution speed of the jobs are given in the following sections.

These notions of internal and external slack time are introduced only in order to reflect the main idea behind MORA and MOTE. Actually, some slack time reclaimed by MOTE can be considered as internal according to our definition. Please keep in mind that we introduced these notions only to emphasize the fact that these two methods do not take advantage from the same kinds of situation.
4.6.2.1 Definitions and notations

MORA modifies the speed of the processors at run-time, during the execution of the jobs. More precisely, MORA bases its decisions to modify the speed of a processor on the current execution context, i.e., the number of jobs that are active, their remaining worst-case execution time, etc. Consequently, in contrast with the energy-aware methods presented up to now, MORA considers that the speed is a notion associated to the jobs, rather than to the processors and it assigns two distinct speeds to every job $\tau_{i,j}$. These two speeds are described below.

1. MORA assigns an execution speed to every job $\tau_{i,j}$ with the interpretation that, whenever $\tau_{i,j}$ is being dispatched to any processor $\pi_k$, the speed of $\pi_k$ is set to the execution speed of $\tau_{i,j}$. Thereby, this notion of job execution speed makes sense only for the active jobs (since inactive jobs cannot be dispatched) and, for sake of clarity, we will prefer the notation $s_j$ instead of $s_{i,j}$ to refer to the execution speed of job $\tau_{i,j}$. No confusion is possible at this stage since it holds for every constrained-deadline task $\tau_i$ that $D_i \leq T_i$, i.e., any job $\tau_{i,j}$ issued from $\tau_i$ is never released before the deadline of the previous job $\tau_{i,j-1}$. Therefore, as long as no deadline is missed, every task $\tau_i$ has at most one active job at a time during the execution of the application. We assume that the execution speed $s_i$ of every active job $\tau_{i,j}$ can
be modified at any time, even during the execution of $\tau_{i,j}$, and the modification is \textit{instantaneously} reflected on the speed of the processor running $\tau_{i,j}$. That is, if the execution speed $s_i$ of the active job $\tau_{i,j}$ is modified while $\tau_{i,j}$ is running on any processor $\pi_k$ then the speed of $\pi_k$ is instantaneously modified as well.

2. The second speed that MORA associates to every active job $\tau_{i,j}$ is noted $s_i^{\text{off}}$ (we omitted the index $j$ of the job in this notation for the same reason as that given above). For every active job $\tau_{i,j}$, $s_i^{\text{off}}$ denotes the \textit{offline speed} of $\tau_{i,j}$ with the interpretation that the execution speed $s_i$ is always set to $s_i^{\text{off}}$ at $\tau_{i,j}$ release time. These offline speeds $s_i^{\text{off}}$ are determined at system design time, i.e., before the system execution, and they are assumed to remain always constant at run-time. These offline speeds can be simply set to the maximal processor speed $s^{\text{max}}$, or they can be determined using a more elaborate offline algorithm. For instance, the offline speed of every job can be set to the identical speed returned by our generic approach presented in Section 4.5.1. Actually, whatever how the offline speeds are determined, they must ensure that all the deadlines are met while the tasks are scheduled upon the $m$ processors under these speeds.

About the power dissipation of the processors, although certain benchmarks provide \textit{average} measured power dissipation, we should not ignore that different functionalities (i.e., different tasks) may have different instruction sequences and therefore require different function units in the processor, thus leading to different consumption profiles. Hence, as already done in [67], we introduce a measurable parameter $e_i$ for each task $\tau_i$. This parameter reflects this difference between the consumption profile of each task and the power dissipation measured while running benchmarks. According to [67], the consumption of any task $\tau_i$ executed for 1 time unit at speed $s$ can be estimated by

$$e_i \cdot (P_{\text{run}}(s) - P_{\text{idle}}) + P_{\text{idle}}$$

where $P_{\text{run}}(s)$ and $P_{\text{idle}}$ are given in the Appendix A (page 347) for different processors. By extension, we denote by $E_i(R, s)$ the energy consumed by the task $\tau_i$ when executed for $R$ time units at speed $s$, i.e.,

$$E_i(R, s) \overset{\text{def}}{=} R \cdot (e_i \cdot (P_{\text{run}}(s) - P_{\text{idle}}) + P_{\text{idle}})$$

As we will see in Section 4.6.2.3, MORA takes this energy consumption function into account in order to improve the energy saving that it provides. This consideration makes MORA very different from the uniprocessor dynamic reclaiming algorithm \textit{DRA}.
CHAPTER 4. EXPLOITING THE DVFS FRAMEWORK

proposed in [7].

MORA is based on reducing at run-time the execution speed of the jobs in order to provide additional energy savings while still meeting all the job deadlines. To achieve this goal, MORA detects whenever some internal slack times can be reclaimed by performing comparisons between the schedule which is actually produced (the actual schedule) and the offline schedule defined below.

Definition 4.7 (The offline schedule [55])

The offline schedule is the schedule in which every job $\tau_{i,j}$ runs at its offline speed $s_{i}^{\text{off}}$ and executes for its WCET $C_i$.

Informally, the definition of the offline schedule can be seen as a refinement of the worst-case schedule. Indeed, the offline schedule is nothing else than the worst-case schedule in which every job is executed at its offline speed. In the remainder of this section, the reader should keep in mind that the algorithm MORA always refers to this offline schedule in order to produce the actual one.

At run-time, whenever any job $\tau_{i,j}$ is dispatched to a processor (say $\pi_k$) in the offline schedule, MORA also dispatches $\tau_{i,j}$ to $\pi_k$ in the actual schedule. Considering only this rule, Figure 4.7a depicts an example of an offline schedule, as well as the notations that will be used throughout this section (these notations are described below). In this picture, a 5-tasks application is executed upon 2 identical processors. Only the first job of each task is represented. The WCET of the tasks are respectively 8, 8, 8, 3 and 5, and we consider a global, work-conserving and FJP scheduler that provides the following priority assignment: $\tau_{1,1} > \tau_{2,1} > \tau_{3,1} > \tau_{4,1} > \tau_{5,1}$. Assuming the same set of tasks as in Figure 4.7a, Figure 4.7b depicts the actual schedule if the actual execution time of the jobs are 3, 2, 5, 1 and 3, respectively. Furthermore, we assume for sake of simplicity that the offline speed $s_{i}^{\text{off}}$ of every job $\tau_{i,1}$ is the maximal processor speed $s_{\text{max}} = 1$. The slack time is represented by the red boxes. During these intervals of time, the concerned processors idle if we consider only the rule introduced above, i.e., the rule following which a job is dispatched to a processor in the actual schedule only if it dispatched to that processor in the offline schedule.

We use the following notations:
4.6 Online slack reclaiming algorithms

- At any time $t$, we denote by $\text{rem}_i(t)$ and $\text{rem}^{\text{off}}_i(t)$ the \textit{remaining} worst-case execution time of the active job $\tau_{i,j}$ at speed $s_{\text{max}}$ in the actual and offline schedule, respectively. For the same reason as that explained earlier, we use the notations $\text{rem}_i(t)$ and $\text{rem}^{\text{off}}_i(t)$ instead of $\text{rem}_{i,j}(t)$ and $\text{rem}^{\text{off}}_{i,j}(t)$ and we will do the same later for other notations. We assume that these quantities $\text{rem}_i(t)$ and $\text{rem}^{\text{off}}_i(t)$ are updated at every time $t$ for every active job $\tau_{i,j}$. For instance in Figure 4.7a at time $t = 3$, we have $\text{rem}^{\text{off}}_1(3) = 5$ while $\text{rem}_1(3) = 0$ (see Figure 4.7b) since $\tau_{1,1}$ completes at time $t = 3$ in the actual schedule. Notice that, from our definition of a processor speed, the \textit{remaining} worst-case execution time of the active job $\tau_{i,j}$ at speed $s$ in the actual and offline schedule is given by $\frac{\text{rem}_i(t)}{s}$ and $\frac{\text{rem}^{\text{off}}_i(t)}{s}$, respectively.

- We denote by $\text{disp}^{\text{off}}_i(t)$ the earliest instant after time $t$ such that $\tau_{i,j}$ is dispatched in...
the offline schedule, considering only the set of jobs that are active at time $t$ in the offline schedule. Informally at the current time $t$, $\text{disp}^{\text{off}}_i(t)$ denotes the next dispatching time in the offline schedule of the active job of $\tau_i$. For instance in Figure 4.7a we have $\text{disp}^{\text{off}}_4(0) = 8$ because the active job $\tau_{4,1}$ is dispatched at time 8 in the offline schedule drawn from this time 0, while considering only the five active jobs (in the offline schedule) at this time 0. In a same way, we have $\text{disp}^{\text{off}}_5(0) = 11$.

Finally, $\text{nextdisp}^{\text{off}}(\pi_k, t)$ denotes the earliest instant after time $t$ at which a job which is active in the actual schedule at time $t$ is dispatched to $\pi_k$ in the offline schedule. Again, only the set of active jobs at time $t$ in the offline schedule are considered in the computation of $\text{nextdisp}^{\text{off}}(\pi_k, t)$. Informally at the current time $t$, $\text{nextdisp}^{\text{off}}(\pi_k, t)$ denotes the next instant in the offline schedule at which any job will be dispatched to $\pi_k$. For instance in Figure 4.7a we have $\text{nextdisp}^{\text{off}}(\pi_2, 2) = 8$ because a job (here, $\tau_{4,1}$) is dispatched at time 8 to processor $\pi_2$ in the offline schedule drawn from this time 2, while considering only the five active jobs (in the offline schedule) at time 2. Similarly, we have $\text{nextdisp}^{\text{off}}(\pi_1, 3) = 8$.

Before going further in the description of MORA, we present in the following section a data structure, named $\alpha$-queue, that stores and updates at run-time a representation of the offline schedule. This data structure is used by MORA in order to compute at any time $t$ the quantities $\text{rem}_i(t)$, $\text{rem}^{\text{off}}_i(t)$, $\text{disp}^{\text{off}}_i(t)$ and $\text{nextdisp}^{\text{off}}(\pi_k, t)$ for every job $\tau_{i,j}$ and every processor $\pi_k$. More precisely, the $\alpha$-queue is used by MORA at run-time for two reasons:

1. in order to know the exact instants at which jobs have to be dispatched to the processors in the actual schedule,
2. in order to know to what extent the execution speed of the active jobs can be reduced.

### 4.6.2.2 The $\alpha$-queue

Since the exact release times of the jobs are unknown while considering sporadic tasks, computing and storing the entire offline schedule cannot be done before the execution of the application. Hence, the $\alpha$-queue only stores and updates at run-time a sufficient part of the offline schedule. This kind of approach (i.e., using a dynamic data structure for embodying a sufficient part of the offline schedule) was previously proposed in [7]. Basically, the $\alpha$-queue is a list that contains, at any time $t$, the worst-case remaining
4.6 Online slack reclaiming algorithms

execution time of every active jobs $\tau_{i,j}$ in the offline schedule, i.e., the quantities $\text{rem}^{\text{off}}_i(t)$. This list is managed according to the following rules. We derived these rules from [7] and we extended them to multiprocessor platforms in [55].

**α-Rule 4.1**

At any time, the $\alpha$-queue is ordered by decreasing job priorities, with the $m$ highest priority jobs at the head of the queue.

**α-Rule 4.2**

Initially the $\alpha$-queue is empty.

**α-Rule 4.3**

Upon the release of a job $\tau_{i,j}$ at time $t$, the WCET $C_i$ of $\tau_{i,j}$ is inserted into the $\alpha$-queue in the correct priority position. This happens only once for each release, no re-insertion at return from preemptions.

**α-Rule 4.4**

As time elapses, the $m$ fields $\text{rem}^{\text{off}}_i(t)$ (if any) at the head of the $\alpha$-queue are decreased with a rate proportional to the offline speeds $s_i^{\text{off}}$. Whenever one field reaches zero, that element is removed and the update continues, still with the $m$ first elements (if any). Obviously, no update is performed when the $\alpha$-queue is empty.

For the same reasons as those explained in [7], the α-Rules 4.1–4.4 make sure that the $\alpha$-queue contains (at any time $t$) only one element for each active job at time $t$ in the offline schedule. Moreover, the $\text{rem}^{\text{off}}_i(t)$ fields reflect the remaining worst-case execution time of every active job $\tau_{i,j}$ at time $t$ in the offline schedule. By consulting the $\alpha$-queue, MORA is therefore able to compute the required information about every active jobs $\tau_{i,j}$ at every time $t$ in the offline schedule, i.e., its remaining worst-case execution time $\text{rem}^{\text{off}}_i(t)$, its next dispatching time $\text{disp}^{\text{off}}_i(t)$ and the next job dispatching time $\text{nextdisp}^{\text{off}}_i(\pi_k, t)$ on every processor $\pi_k$. 

287
CHAPTER 4. EXPLOITING THE DVFS FRAMEWORK

Note that, as explained in [7], the dynamic reduction of the $\text{rem}_i^{\text{off}}(t)$ fields from $\alpha$-Rule 4.4 does not need to be performed at every time unit. Instead, for efficiency, we perform the reduction only before MORA modifies a speed, by taking into account the time elapsed since the last update. Formally, if $\Delta t$ time units elapsed since the last update, the $m$ fields at the head of the $\alpha$-queue are reduced as follows:

$$\text{rem}_i^{\text{off}}(t + \Delta t) \leftarrow \text{rem}_i^{\text{off}}(t) - s_i^{\text{off}} \cdot \Delta t.$$ 

The above approach relies on two facts: as we will see in the next section, the speed adjustment decisions will be taken only at job release time (where the execution speed of the released job is set to its offline speed), at job dispatching time in the offline schedule and whenever a processor completes a job (or idles) in the actual schedule. Hence, it is necessary to have an accurate $\alpha$-queue only at these instants. Second, between these specific instants, each task is effectively executed non-preemptively in the actual schedule.

### 4.6.2.3 Principle of MORA

As explained on page 282, whenever a job is dispatched in the offline schedule to any processor $\pi_k$, it is also dispatched to $\pi_k$ in the actual schedule. However, we will see below that MORA sometimes starts the execution of some jobs earlier in the actual schedule than in the offline one (by profiting from an early job completion). As a result, when a job $\tau_{i,j}$ is dispatched at time $t$ in the offline schedule (and thus also in the actual one if it is not finished yet), its remaining worst-case execution time $\text{rem}_i(t)$ could be lower than $\text{rem}_i^{\text{off}}(t)$ if it was executed earlier in the actual schedule. This situation is illustrated in Figure 4.8, which considers the same set of tasks as in Figure 4.7. At time $t = 2$, $\tau_{2,1}$ completes in the actual schedule on processor $\pi_2$ and leaves 6 unused time units, i.e., 6 units of internal slack time (see Figure 4.8b). In this example, MORA reclaims these 6 time units by starting the execution of $\tau_{5,1}$ (see Figure 4.8c. We will see below how MORA decides on the job that reclaims this slack time. Then, when $\tau_{5,1}$ is dispatched to $\pi_2$ in the offline schedule at time $t = 11$ (see Figure 4.8a), it is also dispatched to $\pi_2$ in the actual one and we have $\text{rem}_{5,1}(11) < \text{rem}_{5,1}^{\text{off}}(11)$. The difference between these remaining worst-case execution times is called the earliness and we denote it by $e_i(t) \overset{\text{def}}{=} \text{rem}_i^{\text{off}}(t) - \text{rem}_i(t)$. According to this earliness, whenever a job $\tau_{i,j}$ is dispatched in both schedules at time $t$, its execution speed $s_i$ may safely be reduced to $s_i'$ so that

$$\frac{\text{rem}_i(t)}{s_i'} = \frac{\text{rem}_i(t) + e_i(t)}{s_i^{\text{off}}}.$$ 

Indeed, under this speed $s_i'$, $\tau_{i,j}$ would complete simultaneously in both schedules if it executes for its WCET (recall that $s_i^{\text{off}}$ remains always constant). Since we know that $\tau_{i,j}$ meets its deadline in the offline schedule, it will also meet its
4.6 Online slack reclaiming algorithms

In the actual schedule, $\tau_{2,1}$ completes earlier because it does not consume its WCET.

MORA profits from the internal slack time left by $\tau_{2,1}$ (more precisely, 6 time units are left by $\tau_{2,1}$) by starting $\tau_{5,1}$ earlier and at a slower speed. This slower speed is such that the remaining worst-case execution time of $\tau_{5,1}$ at time 2 increases by 6 time units.

**Figure 4.8:** Rules 4.1 and 4.2 of MORA.
CHAPTER 4. EXPLOITING THE DVFS FRAMEWORK

Rule 4.1 (From previous work [55])

Whenever any job $\tau_{i,j}$ is dispatched to any processor $\pi_k$ at time $t$ in the offline schedule, $\tau_{i,j}$ is also dispatched to $\pi_k$ at time $t$ in the actual schedule and its execution speed $s_i$ is modified according to

$$s_i \leftarrow \frac{\text{rem}(t) \cdot s_i^{\text{off}}}{\text{rem}^{\text{off}}(t)}$$

(4.8)

The main idea of MORA in order to profit from an early job completion can be summarized as follows: whenever a job completes in the actual schedule without consuming its WCET, the internal slack time left by this job can be reclaimed by starting the execution of any other waiting job earlier; and since the selected waiting job receives extra time for its execution, it can thereby reduce its execution speed. Using this concept, Figure 4.8b shows how MORA takes advantage from an early job completion. When $\tau_{2,1}$ completes at time $t = 2$ in the actual schedule, MORA selects a waiting job (here, $\tau_{5,1}$) and executes it during the 6 time units left by $\tau_{2,1}$. Since $\tau_{5,1}$ is granted to execute for 6 extra time units (in comparison with the offline schedule), MORA reduces its execution speed $s_5$ in such a manner that its remaining worst-case execution time $\text{rem}_5(2)$ increases by 6 time units. The selected job (here, $\tau_{5,1}$) is the one for which the resulting speed reduction leads to the highest energy saving. Formally, MORA selects a waiting job and decreases its execution speed as described by the following Rule 4.2.

Rule 4.2 (From previous work [55])

Whenever a processor $\pi_k$ completes a job at time $t$ (or idles such that at time 0) in the actual schedule, MORA performs the following treatment:

1. Use the $\alpha$-queue to compute the next dispatching time $\text{nextdisp}^{\text{off}}(\pi_k, t)$ on processor $\pi_k$ and perform the steps (a)–(d) for every waiting job $\tau_{i,j}$ at time $t$ in the actual schedule.

(a). Compute the amount $L_i(t)$ of extra time units that $\tau_{i,j}$ could use in the actual schedule if it was dispatched at time $t$, i.e.,

$$L_i(t) \overset{\text{def}}{=} \min \{\text{nextdisp}^{\text{off}}(\pi_k, t), \text{disp}^{\text{off}}_i(t)\} - t$$
4.6 Online slack reclaiming algorithms

For instance in Figure 4.8b we have \( \text{nextdisp}^{\text{off}}(\tau_2, 2) = 8 \) and \( \text{disp}^{\text{off}}(2) = 11 \). Thus, \( L_5(2) = \min[8, 11] - 2 = 6 \).

(b). Compute what would be the resulting execution speed \( s'_i \) if \( \tau_{i,j} \) was granted to execute for these \( L_i(t) \) extra time units. By taking into account the current earliness of \( \tau_{i,j} \), \( s'_i \) must provide

\[
\frac{\text{rem}_i(t)}{s'_i} = \frac{\text{rem}_i(t) + \epsilon_i(t)}{s^{\text{off}}_i} + L_i(t)
\]

leading to

\[
s'_i \overset{\text{def}}{=} \frac{\text{rem}_i(t) \cdot s^{\text{off}}_i}{\text{rem}^{\text{off}}_i(t) + L_i(t) \cdot s^{\text{off}}_i}
\]

(c). Estimate what would be the resulting execution speed \( s''_i \) if \( \tau_{i,j} \) did not received these \( L_i(t) \) extra time units. If we denote by \( t' \geq t \) the instant at which \( \tau_{i,j} \) will be dispatched in the offline schedule, we know by Rule 4.1 that \( s''_i \) will be computed at time \( t' \) as follows:

\[
s''_i \overset{\text{def}}{=} \frac{\text{rem}_i(t') \cdot s^{\text{off}}_i}{\text{rem}^{\text{off}}_i(t')}
\]

Therefore, assuming that \( \tau_{i,j} \) is not executed in the actual schedule until it is dispatched in the offline schedule, it holds at time \( t' \) that \( \text{rem}_i(t') = \text{rem}_i(t) \) and thus,

\[
s''_i \overset{\text{def}}{=} \frac{\text{rem}_i(t) \cdot s^{\text{off}}_i}{\text{rem}^{\text{off}}_i(t)}
\]

(d). Compute the energy saving \( \Delta E_i \) between the complete execution of \( \tau_{i,j} \) at speed \( s''_i \) and at speed \( s'_i \):

\[
\Delta E_i \leftarrow E_i \left( \frac{\text{rem}_i(t)}{s''_i}, s''_i \right) - E_i \left( \frac{\text{rem}_i(t)}{s'_i}, s'_i \right)
\]

2. Dispatch the active job \( \tau_{x,y} \) with the largest \( \Delta E_x \) to processor \( \pi_k \). If \( \Delta E_{i,j} \leq 0 \) for all the waiting jobs \( \tau_{i,j} \), then dispatch the waiting job (if any) with the highest priority in order to complete it earlier and to potentially increase the length of the future internal slack time.
3. If there is a selected job $\tau_{x,y}$, set its execution speed $s_x$ to the computed speed $s'_x$. Otherwise, idle the processor $\pi_k$.

Notice that Rule 4.1 takes priority over Rule 4.2, meaning that, if a processor $\pi_k$ completes a job in the actual schedule exactly when a job is dispatched to $\pi_k$ in the offline one, only Rule 4.1 is performed. Algorithm 8 provides a pseudo-code of MORA and we demonstrate its correctness in the following section.

4.6.2.4 Correctness of MORA

In Lemma 4.4 below, we formally prove that MORA does not jeopardize the schedulability of the application. The following Lemmas 4.2 and 4.3 are just some base results of Lemma 4.4.

**Lemma 4.2 (From previous work [56])**

Let $\tau$ be any set of sporadic and constrained-deadline tasks and let $S$ be any global, preemptive, work-conserving and FJP scheduler according to the definitions given in Section 1.3.5 (on page 29). Suppose that $\tau$ is scheduled by $S$ while using MORA, if at any time $t \geq 0$ it holds for every active jobs $\tau_{i,j}$ that

$$\forall 0 \leq t' \leq t : \text{rem}_i(t') \leq \text{rem}^{\text{off}}_i(t')$$

then $\exists t', 0 \leq t' \leq t$, such that $\exists \tau_{i,j}$ running at time $t'$ in the offline schedule and waiting at time $t'$ in the actual schedule.

**Proof**

The proof is made by contradiction. Suppose that there exists a time $t'$ ($0 \leq t' \leq t$) such that $\exists \tau_{i,j}$ running at time $t'$ in the offline schedule and waiting in the actual schedule. Since $\tau_{i,j}$ is running in the offline schedule, there can be at most $(m - 1)$ jobs with a higher priority than $\tau_{i,j}$ in the offline schedule. On the other hand, since $\tau_{i,j}$ is waiting in the actual schedule, there must be at least $m$ jobs with a higher priority than $\tau_{i,j}$ in the actual schedule. Consequently, there is at least one job (say $\tau_{x,y}$) at time $t'$ with a higher priority than $\tau_{i,j}$ in both schedules and such that $\tau_{x,y}$ is completed in the offline schedule and still running in the actual one. For this job, it therefore holds that $\text{rem}_x(t') > \text{rem}^{\text{off}}_x(t')$, leading to contradict our hypothesis.
4.6 Online slack reclaiming algorithms

Algorithm 8: MORA algorithm

begin
  // Initialization step
  Determine the offline speed $s_i^{\text{off}}$ of every task $\tau_i$;
  $\alpha$-queue ← φ;

  At job release (say $\tau_{i,j}$) at time $t$:
  Update the $\alpha$-queue according to $\alpha$-Rule 4.4;
  Insert the value of $C_i$ into the $\alpha$-queue according to $\alpha$-Rule 4.3;
  $s_i ← s_i^{\text{off}}$;
end procedure

When any job $\tau_{i,j}$ is dispatched to any processor $\pi_k$ in the offline schedule at time $t$:
  Update the $\alpha$-queue according to $\alpha$-Rule 4.4;
  if (a job $\tau_{x,y} \neq \tau_{i,j}$ is running on $\pi_k$) then Preempt $\tau_{x,y}$;
  $s_i ← \frac{\text{rem}(t)}{\text{rem}(t) + L_i} \cdot s_i^{\text{off}}$;
end procedure

When any processor $\pi_k$ is getting idle at time $t$:
  Update the $\alpha$-queue according to $\alpha$-Rule 4.4;
  nextdisp$^{\text{off}}_k ← \alpha$-queue.computeNextDisp($\pi_k$, $t$);
  // for each waiting job at time $t$ in the actual schedule
  foreach $\tau_{i,j} \in \text{active}(\tau, t)$ do
    disp$^{\text{off}}_i ← \alpha$-queue.computeDisp($\tau_{i,j}, t$);
    $L_i ← \min\{\text{nextdisp}^{\text{off}}_k, \text{disp}^{\text{off}}_i\} - t$;
    $s_i' ← \frac{\text{rem}(t)}{\text{rem}(t) + L_i} \cdot s_i^{\text{off}}$;
    $s_i'' ← \frac{\text{rem}(t)}{\text{rem}(t) + L_i} \cdot s_i^{\text{off}}$;
    $\Delta E_i ← E_i\left(\frac{\text{rem}(t)}{s_i''}, s_i''\right) - E_i\left(\frac{\text{rem}(t)}{s_i'}, s_i'\right)$;
  end foreach
  if $\Delta E_i ≤ 0 \forall \tau_{i,j} \in \text{active}(\tau, t)$ then
    $\tau_{x,y} ← \text{job with the highest priority}$;
  else
    $\tau_{x,y} ← \text{job with the largest } \Delta E_i$;
  end if
  if $\tau_{x,y} \neq \phi$ then
    $s_x ← s_x'$;
    Dispatch $\tau_{x,y}$ to processor $\pi_k$;
  else
    Idle the processor $\pi_k$;
  end if
end procedure

end
Lemma 4.3 (From previous work [56])

Let \( \tau \) be any set of sporadic and constrained-deadline tasks and let \( S \) be any global, preemptive and FJP scheduler according to the definitions given in Section 1.3.5 (page 29). Suppose that \( \tau \) is scheduled by \( S \) while using MORA. If at time \( t \geq 0 \) it holds for every active jobs \( \tau_{i,j} \) that

\[
\forall 0 \leq t' \leq t : \text{rem}_i(t') \leq \text{rem}^{\text{off}}_i(t')
\]

then \( \exists t', 0 \leq t' \leq t \), such that \( \exists \tau_{i,j} \) running at time \( t' \) in the offline schedule and such that the last speed modification of \( \tau_{i,j} \) was performed according to Rule 4.2.

Proof

The proof is made by contradiction. Suppose that there exists a time \( t' \) (\( 0 \leq t' \leq t \)) such that \( \exists \tau_{i,j} \) running at time \( t' \) in the offline schedule and such that the last modification of \( s_i \) was performed according to Rule 4.2. Let \( t_{\text{active}} \) and \( t_{\text{off}} \) be the largest instants before time \( t' \) at which \( \tau_{i,j} \) was dispatched in the actual and offline schedule, respectively. Notice that the case where \( \tau_{i,j} \) is not dispatched before time \( t' \) in the actual schedule leads to a contradiction of Lemma 4.2. Therefore, only two cases can occur:

1. \( t_{\text{active}} \leq t_{\text{off}} \). In this case, \( s_i \) would have been modified at time \( t_{\text{off}} \) according to Rule 4.1, leading to contradict our hypothesis.
2. \( t_{\text{active}} > t_{\text{off}} \). This directly leads to a contradiction of Lemma 4.2 since it means that at time \( t_{\text{off}} \leq t' \), \( \tau_{i,j} \) is running in the offline schedule while it is waiting in the actual one.

The lemma follows from these two contradictions.

Lemma 4.4 (From previous work [56])

Let \( S \) be any global, preemptive and FJP scheduler according to the definitions given in Section 1.3.5 (page 29) and let \( \tau \) be any set of sporadic and constrained-deadline tasks that is schedulable by \( S \) when every job \( \tau_{i,j} \) is executed at its offline speed \( s_{i,\text{off}} \). Then, every job deadline is still met when the application is scheduled by \( S \) while using MORA.

Proof

The proof consists in showing that \( \forall \tau_{i,j} \) we have
rem_i(d_{i,j}) \leq \text{rem}_i^{off}(d_{i,j}) \tag{4.9}

while using MORA. Indeed, since the offline schedule meets all the job deadlines, we have \( \text{rem}_i^{off}(d_{i,j}) = 0 \) \( \forall \tau_{i,j} \). Therefore, having \( \text{rem}_i(d_{i,j}) \leq \text{rem}_i^{off}(d_{i,j}) \) leads to \( \text{rem}_i(d_{i,j}) = 0 \) \( \forall \tau_{i,j} \), meaning that the actual schedule also meets all the deadlines. The proof is made by induction on the time.

**Base case.** Initially, at time \( t = 0 \), it obviously holds that \( \text{rem}_i(0) = \text{rem}_i^{off}(0) \) \( \forall \tau_{i,j} \).

**Inductive step.** Let \( t > 0 \) be any instant during the execution of the application and suppose that \( \forall \tau_{i,j} \) and \( 0 \leq t' \leq t \) it holds that \( \text{rem}_i(t') \leq \text{rem}_i^{off}(t') \). We prove in the following that this yields

\[
\forall \tau_{i,j} : \text{rem}_i(\text{next}(t)) \leq \text{rem}_i^{off}(\text{next}(t)) \tag{4.10}
\]

where next\( t \) denotes the earliest instant after time \( t \) such that one (or more) of the following events occurs: release of a job, deadline of a job, completion of a job in the actual or offline schedule or dispatching of a job in the actual or offline schedule. Obviously, if Inequality 4.10 holds then Inequality 4.9 also holds since next\( t \) can denote every job deadline.

From the definition of next\( t \), it holds that in both schedules every processor is either idle or executes one and only one job in the time interval \([t, \text{next}(t)]\). Therefore, the state (waiting or running) of every active job does not change within \([t, \text{next}(t)]\) in both schedule, leading to the three following properties:

**Prop 1.** At time \( t \), it holds for every waiting job \( \tau_{i,j} \) in the actual schedule that

\[
\text{rem}_i(\text{next}(t)) = \text{rem}_i(t)
\]

**Prop 2.** At time \( t \), it holds for every waiting job \( \tau_{i,j} \) in the offline schedule that

\[
\text{rem}_i^{off}(\text{next}(t)) = \text{rem}_i^{off}(t)
\]

**Prop 3.** At time \( t \), it holds for every running job \( \tau_{i,j} \) in the actual schedule that

\[
\text{rem}_i(\text{next}(t)) \leq \text{rem}_i(t)
\]
The first part of the proof shows that Inequality 4.10 holds for every waiting job at time \( t \) in the actual schedule and the second part shows that it also holds for every running job at time \( t \) in the actual schedule. Therefore, we will be in a position to conclude that Inequality 4.10 holds for every active job in the actual schedule and thus for every released job during the execution of the application.

**Part 1.** Let \( \tau_{i,j} \) be any waiting job at time \( t \) in the actual schedule. From Lemma 4.2, we know that \( \tau_{i,j} \) is also waiting in the offline schedule and since by hypothesis \( \text{rem}_i(t) \leq \text{rem}^{\text{off}}_i(t) \), we know from Properties 1 and 2 that

\[
\text{rem}_i(\text{next}(t)) \leq \text{rem}^{\text{off}}_i(\text{next}(t))
\]

**Part 2.** Let \( \tau_{i,j} \) be any running job at time \( t \) in the actual schedule. Regarding its execution speed \( s_i \), only two cases can occur: its last modification was performed by Rule 4.1 (case 1) or by Rule 4.2 (case 2).

**Part 2: Case 1.** \( \tau_{i,j} \) is running at time \( t \) in the actual schedule and the last modification of \( s_i \) was performed according to Rule 4.1, i.e., when it was dispatched in the offline schedule (say at time \( t_{\text{off}} \leq t \)). From Lemma 4.2, we know that \( \tau_{i,j} \) is also running at time \( t \) in the offline schedule. Therefore, it holds that \( \tau_{i,j} \) is running at both instants \( t_{\text{off}} \) and \( t \) in both schedules and we can conclude that \( \tau_{i,j} \) is executed non-preemptively in both schedules within \( [t_{\text{off}}, t] \). Indeed, \( \tau_{i,j} \) cannot be preempted in the actual schedule without being preempted in the offline schedule and inversely, if it was preempted by another job in the offline schedule, it would have been also preempted by the same job in the actual one according to Rule 4.1. But since \( \tau_{i,j} \) is running at time \( t \) in the actual schedule, it means that this other job would have completed and \( \tau_{i,j} \) re-dispatched before time \( t \). Thus, the speed \( s_i \) of \( \tau_{i,j} \) would have been modified according to Rule 4.2 at its re-dispatching time if it was preempted during \( [t_{\text{off}}, t] \). As a result, \( \tau_{i,j} \) is executed non-preemptively in both schedules within \( [t_{\text{off}}, t] \) and it holds that

\[
\text{rem}_i(\text{next}(t)) = \text{rem}_i(t_{\text{off}}) - s_i \cdot (\text{next}(t) - t_{\text{off}}) \quad (4.11)
\]

and

\[
\text{rem}^{\text{off}}_i(\text{next}(t)) = \text{rem}^{\text{off}}_i(t_{\text{off}}) - s_i^{\text{off}} \cdot (\text{next}(t) - t_{\text{off}}) \quad (4.12)
\]

After the speed modification by Rule 4.1 at time \( t_{\text{off}} \), we know that

\[
s_i = \frac{\text{rem}_i(t_{\text{off}})}{\text{rem}^{\text{off}}_i(t_{\text{off}})} \cdot s_i^{\text{off}}
\]
and Equality 4.11 can be rewritten as

\[ \text{rem}_i(\text{next}(t)) = \text{rem}_i(t_{\text{off}}) - \frac{\text{rem}_i(t_{\text{off}})}{\text{rem}_i(t_{\text{off}})} \cdot s_i \cdot (\text{next}(t) - t_{\text{off}}) \]

Multiplying the above equality by \( \frac{\text{rem}_i(t_{\text{off}})}{\text{rem}_i(t_{\text{off}})} \) yields

\[ \text{rem}_i(\text{next}(t)) \cdot \frac{\text{rem}_i(t_{\text{off}})}{\text{rem}_i(t_{\text{off}})} = \text{rem}_i(t_{\text{off}}) - s_i \cdot (\text{next}(t) - t_{\text{off}}) \]

and since the right-hand side of this equality corresponds to that of Equality 4.12, it holds that

\[ \text{rem}_i(\text{next}(t)) \cdot \frac{\text{rem}_i(t_{\text{off}})}{\text{rem}_i(t_{\text{off}})} = \text{rem}_i(t_{\text{off}}) - s_i \cdot (\text{next}(t) - t_{\text{off}}) \]

Since by hypothesis \( \text{rem}_i(t_{\text{off}}) \leq \text{rem}_i(t_{\text{off}}) \), we have \( \frac{\text{rem}_i(t_{\text{off}})}{\text{rem}_i(t_{\text{off}})} \geq 1 \) and thus,

\[ \text{rem}_i(\text{next}(t)) \leq \text{rem}_i(t_{\text{off}}) \]

**Part 2: Case 2.** \( \tau_{i,j} \) is running at time \( t \) in the actual schedule and the last modification of \( s_i \) was performed according to Rule 4.2. Therefore, we know from Lemma 4.3 that \( \tau_{i,j} \) is waiting at time \( t \) in the offline schedule and from Properties 2 and 3,

1. \( \text{rem}_i(t_{\text{off}}) = \text{rem}_i(t_{\text{off}}) \) (from Property 2), and
2. \( \text{rem}_i(t_{\text{off}}) \leq \text{rem}_i(t_{\text{off}}) \) (from Property 3).

Since by hypothesis \( \text{rem}_i(t_{\text{off}}) \leq \text{rem}_i(t_{\text{off}}) \), it holds from the two above expressions that

\[ \text{rem}_i(\text{next}(t)) \leq \text{rem}_i(t_{\text{off}}) \]

The theorem follows from the enumeration of these two cases.

### 4.6.3 Multiprocessor One Task Extension (MOTE)

Algorithm MOTE, which stands for “Multiprocessor One Task Extension”, is a multiprocessor extension of the algorithm proposed in [62] and usually referred to as OTE. The aim of MOTE is to further improve the energy saving provided by our offline strategies described in the previous section. As for MORA, we assume that the processors are inde-
pendent (according to Definition 4.2 on page 246) and thus, every processor can change its speed at any time independently from the speed of the other processors. Nevertheless, we still assume that all the processors share the same minimal and maximal speeds $s^{\text{min}}$ and $s^{\text{max}}$, and the execution speed of any processor is always set to $s^{\text{min}}$ when it idles.

Basically, MOTE is a low-complexity online algorithm that aims to further reduce the speeds of the CPUs by performing “local” adjustments at run-time. MOTE can be used in combination with MORA and it is designed to reclaim the external slack time. Figure 4.9 depicts an extremely simple example that perfectly reflects the main idea of MOTE. In this figure, there is a single job (i.e., $\tau_{i,j}$) waiting for execution at time $t$. The down-arrow labeled $d_{i,j}$ represents its deadline and the unlabeled up-arrows represent the earliest possible release times of other jobs (one of them could be the next job of $\tau_i$). The length of the box represents the WCET of $\tau_{i,j}$. In this simple example, one can easily see that after the execution of $\tau_{i,j}$, the processor will unavoidably idle until the next job release (even if $\tau_{i,j}$ executes for its WCET). The green box represents the external slack time. In this scenario, MORA cannot reduce the speed of $\tau_{i,j}$ at time $t$ since MORA is designed in such a manner that it profits from another kind of situation, i.e., it executes jobs earlier (and at a lower speed) by profiting from the internal slack time left by other jobs. On the contrary, MOTE is designed in such a manner that at time $t$, it detects that the processor will idle as soon as $\tau_{i,j}$ will be completed. Thus, MOTE estimates the length of this future external slack time and reduces the execution speed of $\tau_{i,j}$ at time $t$ accordingly.

![Figure 4.9: At time $t$, MOTE detects the unavoidable external slack time after the execution of $\tau_{i,j}$ (represented by the green area) and it slows down the execution speed of $\tau_{i,j}$ accordingly.](image)

1 Another option would have been to set the processor into one of its sleep modes as soon as it starts idling. However, the operation of entering/exiting a sleep mode requires a certain amount of time and energy and the appropriate sleep mode should be selected according to the duration of the idle period. For a long idle period, it could even be advantageous in term of consumption to totally turn off the processor. For sake of simplicity, we chose not to deal with the processor sleep modes.

298
The combination of MORA and MOTE will be addressed in the next section. In the following, we assume that MOTE is used without any other energy-aware mechanism.

### 4.6.3.1 Definitions and notations

Before describing the algorithm MOTE, let us introduce the following notations and definitions.

**Definition 4.8 (Last release time of a task [57])**

At any time $t$ during the execution of the application, the last release time $\text{LastRel}_i(t)$ of any task $\tau_i$ is the latest instant $t' \leq t$ at which $\tau_i$ released a job. We assume that $\text{LastRel}_i(t)$ is updated at run-time whenever $\tau_i$ releases a job and $\text{LastRel}_i(0)$ is initially set to $-T_i$ (see Equation 4.13 to understand this initialization).

**Definition 4.9 (The “potential release” function of a task [57])**

At any time $t$ and for any instant $t' \geq t$, the potential release function $\text{PotRel}_i(t,t')$ of any task $\tau_i$ indicates whether $\tau_i$ could release a job at time $t'$. In the sporadic task model, the period $T_i$ denotes the minimal inter-arrival delay between two consecutive job releases of $\tau_i$ and $\text{PotRel}_i(t,t')$ is therefore given by

$$\text{PotRel}_i(t,t') \overset{\text{def}}{=} \begin{cases} 1 & \text{if } t' \geq \text{LastRel}_i(t) + T_i \\ 0 & \text{otherwise} \end{cases}$$

(4.13)

Notice that $\text{LastRel}_i(0)$ is initially set to $-T_i$ so that $\text{PotRel}_i(0,t') = 1 \forall t'$. Indeed, the sporadic task model considers that any task can release its first job at any instant $t' \geq 0$.

The proposed method MOTE assigns an execution speed to every job $\tau_{i,j}$ with the same interpretation as that of the job execution speed in algorithm MORA. That is, whenever $\tau_{i,j}$ is being dispatched to any processor $\pi_k$, the speed of processor $\pi_k$ is set to the execution speed of $\tau_{i,j}$. Once again, this notion of job execution speed makes sense only for the active jobs (since inactive jobs cannot be dispatched) and we will use the notation $s_j$ instead of $s_{i,j}$ to refer to the execution speed of any job $\tau_{i,j}$. No confusion is possible since, for any constrained-deadline task $\tau_i$, it holds that $D_i \leq T_i$, i.e., any job
\( \tau_{i,j} \) is never released before the deadline of the previous job \( \tau_{i,j-1} \). Therefore, as long as no deadline is missed, every task \( \tau_i \) has at most one active job at any time \( t \) during the execution of the application.

**Definition 4.10 (Remaining WCET of a job at speed \( s \))**

At any time \( t \), we denote by \( \text{rem}_i(t) \) the remaining worst-case execution time of the last released job of task \( \tau_i \) if executed at maximal processor speed \( s_{\text{max}} \). If \( \tau_i \) has no active job at time \( t \) then \( \text{rem}_i(t) \) is simply assumed to be 0. By extension, we denote by \( \text{rem}_i^s(t) \) the remaining worst-case execution time of the last released job of task \( \tau_i \) if executed at speed \( s \), i.e., \( \text{rem}_i^s(t) = \frac{\text{rem}_i(t)}{s} \) according to our interpretation of the processor speed.

**Definition 4.11 (The “potentially active” function of a task [57])**

At any time \( t \) during the execution of the application and for any instant \( t' \geq t \), the potentially active function \( \text{PotAct}_i(t,t') \) indicates whether \( \tau_i \) has an active job at time \( t \) that could be still active at time \( t' \). That is, this function returns 1 only if \( \tau_i \) has an active job at time \( t \) and \( t' \) is not larger than the deadline of this active job, i.e.,

\[
\text{PotAct}_i(t,t') \overset{\text{def}}{=} \begin{cases} 1 & \text{if } \text{rem}_i(t) > 0 \text{ and } t \leq t' < \text{LastRel}_i(t) + D_i \\ 0 & \text{otherwise} \end{cases}
\]

**Lemma 4.5**

At any time \( t \), the following function \( \Pi_i(t,t') \), when non-negative, provides a lower bound on the number of processors that will idle at time \( t' \geq t \), while ignoring the schedule of the active job \( \tau_{i,j} \) (if any). \( \Pi_i(t,t') \) is given by

\[
\Pi_i(t,t') \overset{\text{def}}{=} m - \sum_{\tau_k \in \tau_i \setminus \{\tau_i\}} \text{PotAct}_k(t,t') - \sum_{\tau_k \in \tau_i} \text{PotRel}_k(t,t') \tag{4.14}
\]

**Proof**

Since the schedulability of the application ensures that every job completes by its deadline, the latest time at which a job \( \tau_{i,j} \) can complete is its absolute deadline \( d_{i,j} \). Moreover, it holds for any value of \( t \) that, \( \forall t' \geq t \) the function
4.6 Online slack reclaiming algorithms

\[ \sum_{\tau_k \in \{\tau_i\}} \text{PotAct}_k(t,t') \]

is decreased only at each time \( t' \) that corresponds to the deadline of any job \( \tau_{k,x} \neq \tau_{i,j} \) active at time \( t \). That is, considering only the active jobs different from \( \tau_{i,j} \) at time \( t \), this function \( \sum_{\tau_k \in \{\tau_i\}} \text{PotAct}_k(t,t') \) provides an upper-bound on the number of active jobs at any time \( t' \geq t \). It therefore holds that the function

\[(m - \sum_{\tau_k \in \{\tau_i\}} \text{PotAct}_k(t,t'))\]

provides a lower bound on the number of idle processors at any time \( t' \geq t \), when only the active jobs different from \( \tau_{i,j} \) at time \( t \) are considered. Then, the term "\( \sum_{\tau_k \in \{\tau_i\}} \text{PotRel}_k(t,t') \)" allows Expression 4.14 to take into consideration the jobs that will be released after time \( t \). Indeed, every job released after time \( t \) also requests a processor and consequently, the lower-bound on the number of idle processors at any time \( t' \geq t \) has to be decreased at each next (possible) release time. The value of the function \( \sum_{\tau_k \in \{\tau_i\}} \text{PotRel}_k(t,t') \) takes this into account, because it is increased by one at each earliest instant at which a task could release a job. Thus, for any instant \( t' \geq t \), this function

\[ \sum_{\tau_k \in \{\tau_i\}} \text{PotRel}_k(t,t') \]

provides the maximal number of jobs that could be released after time \( t \) and still active at time \( t' \geq t \). The difference \( \Pi_i(t,t') \) between the minimum number of idle processors at time \( t' \) (considering only the active jobs different from \( \tau_{i,j} \) at the current time \( t \)) and the maximum number of jobs that could be released after time \( t \) and still active at time \( t' \) provides a lower bound on the number of idle processors at time \( t' \) (without considering the execution of \( \tau_{i,j} \) at time \( t \)); if this difference is negative, it can be the case where no processor idles at time \( t' \).

**Lemma 4.6**

Whenever any job \( \tau_{i,j} \) is dispatched to any CPU \( \pi_k \) at time \( t \), the earliest future instant in the schedule at which another job (possibly from the same task) could have no other choice than to be dispatched to \( \pi_k \) is given by:
CHAPTER 4. EXPLOITING THE DVFS FRAMEWORK

\[ t_{next} \overset{\text{def}}{=} \begin{cases} \min \{t' \geq t \mid \Pi_i(t, t') \leq 0\} & \text{if } m \leq n \\ +\infty & \text{otherwise} \end{cases} \]

Proof

The proof is a direct consequence of Lemma 4.5. Indeed, within the time interval \([t, t_{next}]\), there is at least one idle processor for every active job different from \(\tau_{i,j}\) (including those which could be released in \([t, t_{next}]\)). That is, every active job in the time interval \([t, t_{next}]\) can be directly dispatched to a processor. Until time \(t_{next}\), we can assume that the processors upon which the active jobs are dispatched are different from \(\pi_k\), but at time \(t_{next}\), every processor (including \(\pi_k\)) could be required to run an active job. Notice that, since job parallelism is forbidden, there will be always at least one idle processor if the number of processors is greater than the number of tasks.

4.6.3.2 MOTE scheme

The main idea behind MOTE is the following: the execution speed of any job can safely be reduced if this speed reduction does not change anything with respect to the schedule of the other jobs. Indeed, since we consider multiprocessor platforms we know that we have to be very careful to any change in the original schedule because of scheduling anomalies (see Chapter 1 Section 1.3.5.4). Hence, our online energy-aware mechanism MOTE focuses only on the last dispatched job and avoids to change the schedule of all the other jobs.

For instance, Figures 4.10a and 4.10b depict a situation where a speed modification is not allowed. These figures represent the schedule of the four first jobs issued from four distinct tasks upon 2 independent DFVS-identical processors. The scheduler is assumed to be FJP and the priority assignment is \(\tau_{4,1} > \tau_{2,1} > \tau_{3,1} > \tau_{4,1}\). Assuming that the four jobs are executed at maximum speed \(s_{\text{max}}\), Figure 4.10a shows that no deadline is missed. Then, on Figure 4.10b, we assume that the execution speed of \(\tau_{2,1}\) has been slowed down, thus prolonging the execution time of \(\tau_{2,1}\). This speed modification delays the execution of \(\tau_{3,1}\) and, if \(\tau_{2,1}\) comes to execute for its WCET, \(\tau_{3,1}\) is dispatched to \(\pi_1\) at time 8 instead of time 6 (that is, the original schedule is modified). As we can see, this causes \(\tau_{3,1}\) to miss its deadline at time 13 if \(\tau_{3,1}\) also executes for its WCET.
4.6 Online slack reclaiming algorithms

On the other hand, Figures 4.11a and 4.11b depict a situation where a speed modification is allowed. These figures show the schedule of 3 sporadic and constrained-deadline tasks executed upon 3 independent DVFS-identical processors. This example is not really pertinent in practice since it considers a very particular case where there are as many tasks as processors. However, it perfectly reflects the main idea of the MOTE algorithm. In these two schedules, \( t = 0 \) is the current time, \( \tau_{1,1}, \tau_{2,1} \) and \( \tau_{3,1} \) are the three active jobs at time 0 and the up and down arrows represent the earliest next release times and the deadline of each task, respectively (the up-arrows are put into brackets since tasks are sporadic and the exact instants of their job releases are therefore unknown). Suppose that at time \( t = 0 \), \( \tau_{1,1} \) and \( \tau_{2,1} \) are dispatched to \( \pi_3 \) and \( \pi_2 \), respectively. Still at this time \( t = 0 \), \( \tau_{3,1} \) is dispatched to the processor \( \pi_1 \) and we can see that \( \pi_1 \) cannot be required by another job than \( \tau_{3,1} \) until (at least) time \( a_{3,1} + T_3 \). Indeed, the future jobs \( \tau_{1,2} \) and \( \tau_{2,2} \) can be released at any time after (or at) the instants \( a_{1,1} + T_1 \) and \( a_{2,1} + T_2 \), respectively. But since the schedulability of the application ensures that \( \tau_{1,1} \) and \( \tau_{2,1} \) will complete.
by their respective deadline $d_{1,1}$ and $d_{2,1}$, if one of them (or both) is released before time $a_{3,1} + T_3$, it (they) can be dispatched either to processors $\pi_3$ or $\pi_2$ (thus leaving $\pi_1$ idle). However from time $a_{3,1} + T_3$, the three jobs $\tau_{1,2}, \tau_{2,2}$ and $\tau_{3,2}$ could be active simultaneously and the processor $\pi_1$ could therefore be required to execute one of them (since the scheduler is assumed to be work-conserving). That is, while ignoring the job $\tau_{3,1}$, the instant $a_{3,1} + T_3$ is the earliest instant after time $t = 0$ at which processor $\pi_1$ could be required.
Mathematically, the instant \( t' = a_{3,1} + T_3 \) is the earliest instant after time \( t = 0 \) such that the function \( \Pi_3(t, t') \) becomes null or negative. Indeed, \( \Pi_3(0, a_{3,1} + T_3) = 3 - 0 - 3 = 0 \) whereas \( \Pi_3(0, t') > 0 \ \forall t' \in [0, a_{3,1} + T_3) \). We thus have \( t_{next} = a_{3,1} + T_3 \) and since processor \( \pi_1 \) will not be required by another job than \( \tau_{3,1} \) until this time \( t_{next} \), the execution speed of \( \tau_{3,1} \) can be reduced so that it will complete at most by time \( \min\{t_{next}, d_{3,1}\} \). The schedule resulting from this speed reduction is depicted in Figure 4.11b. To provide the reader with a good understanding of the function \( \Pi \), Figure 4.12 depicts the evolution of the functions \( \sum_{\tau_i \in T} \text{PotAct}_k(0, t') \), \( \sum_{\tau_i \in T} \text{PotRel}_k(0, t') \) and \( \Pi_3(0, t') \) for all time \( t' \) such that \( 0 \leq t' \leq a_{3,1} + T_3 \).

**Principle of MOTE:** MOTE reduces the speed of a processor only at job dispatching time. Suppose that job \( \tau_{i,j} \) is dispatched to processor \( \pi_\ell \) at time \( t \) during the execution of the application. Since we consider FJP schedulers, this time \( t \) is either the release time of \( \tau_{i,j} \) or the completion time of a higher priority job. At this time, MOTE dispatches the job to \( \pi_\ell \) and determines the earliest instant \( t_{next} > t \) such that \( \pi_\ell \) could be required to execute another job than \( \tau_{i,j} \). For instance in Figure 4.11, \( t_{next} \) is the instant \( a_{3,1} + T_3 \). To compute \( t_{next} \), MOTE uses of the function \( \Pi_i(t, t') \). As showed in Figure 4.12, this function varies only at job deadlines and at next (earliest) release times; Between these instants, the function remains constant. Therefore, finding the earliest instant \( t_{next} \) such that \( t_{next} > t \) and \( \Pi_i(t, t_{next}) \leq 0 \) can be achieved by scanning the deadline and the (earliest) future release time of every task. It follows from Lemma 4.6 that \( \pi_\ell \) will not execute another job than \( \tau_{i,j} \) until (at least) this instant \( t_{next} \). As a consequence, if such an instant \( t_{next} \) is found then the execution speed of \( \tau_{i,j} \) can be safely reduced at time \( t \) in such a manner that \( \tau_{i,j} \) will complete at most by time \( \min\{d_{i,j}, t_{next}\} \) (obviously, the execution speed of a processor can never be reduced below \( s_{\text{min}} \)). This manner of reducing the speed is formally expressed in Lemma 4.8 given below. But let us first introduce the following result.
CHAPTER 4. EXPLOITING THE DVFS FRAMEWORK

(a) Evolution of the function \( \sum_{\tau_k \in \tau \setminus \{\tau_3\}} \text{PotAct}_k(0, t') \) for all \( 0 \leq t' \leq a_{3,1} + T_3 \).

(b) Evolution of the function \( \sum_{\tau_k \in \tau} \text{PotRel}_k(0, t') \) for all \( 0 \leq t' \leq a_{3,1} + T_3 \).

(c) Evolution of the function \( \Pi_3(0, t') \) for all \( 0 \leq t' \leq a_{3,1} + T_3 \). Recall that here, \( \Pi_3(0, t') = 3 - \sum_{\tau_k \in \tau \setminus \{\tau_3\}} \text{PotAct}_k(0, t') - \sum_{\tau_k \in \tau} \text{PotRel}_k(0, t') \).

Figure 4.12: Illustration of the functions \( \sum_{\tau_k \in \tau \setminus \{\tau_3\}} \text{PotAct}_k(0, t'), \sum_{\tau_k \in \tau} \text{PotRel}_k(0, t') \) and \( \Pi_3(0, t'), \forall t' \in [0, a_{3,1} + T_3] \).
4.6 Online slack reclaiming algorithms

**Lemma 4.7**

If $\tau_{i,j}$ denotes any job with a remaining WCET of $\text{rem}_i^j(t)$ at time $t$ while executed at speed $s_i$, then the speed $s_i'$ such that its remaining WCET at time $t$ becomes $R$ is given by

$$s_i' = \frac{\text{rem}_i^j(t)}{R} \cdot s_i$$  \hspace{1cm} (4.15)

**Proof**

By definition of the speed, we have

$$\text{rem}_i^j(t) = \frac{\text{rem}_i(t)}{s_i}$$

Replacing $s_i'$ in the right-hand side leads to

$$\text{rem}_i^j(t) = \frac{\text{rem}_i(t)}{\text{rem}_i^j(t) \cdot s_i} = \frac{\text{rem}_i(t) \cdot R}{\text{rem}_i^j(t) \cdot s_i}$$

Then, replacing $\text{rem}_i^j(t)$ with $\frac{\text{rem}_i(t)}{s_i}$ in the right-hand side yields

$$\text{rem}_i^j(t) = \frac{\text{rem}_i(t) \cdot R}{\text{rem}_i(t) \cdot s_i} = R$$

The lemma follows.

**Lemma 4.8**

Let $\tau_{i,j}$ denote any job with a remaining WCET of $\text{rem}_i^j(t)$ at time $t$ while executed at speed $s_i$. Assuming that $\tau_{i,j}$ is dispatched to any processor at time $t$, the execution speed $s_i'$ such that $\tau_{i,j}$ will complete at most at time $\min\{d_{i,j}, t_{\text{next}}\}$ is given by

$$s_i' = \frac{\text{rem}_i^j(t) \cdot s_i}{\min\{d_{i,j}, t_{\text{next}}\} - t}$$

**Proof**
CHAPTER 4. EXPLOITING THE DVFS FRAMEWORK

The proof is a direct consequence of Lemma 4.7 and is obtained simply by replacing $R$ in Expression 4.15 with the available time within $\left[ t, \min \{D_{ij}, t_{\text{next}}\} \right]$, i.e.,

$$\min \{D_{ij}, t_{\text{next}}\} - t.$$  

Finally, MOTE works as follows. Whenever any job $\tau_{i,j}$ is dispatched to any CPU $\pi_k$ during the execution of the application, MOTE determines the earliest instant $t_{\text{next}}$ such that $\Pi_i(t, t_{\text{next}}) \leq 0$ and, if $t_{\text{next}} > t$ then the execution speed of $\tau_{i,j}$ is modified as follows:

$$s_i \leftarrow \max \left\{ s_i^\min, \min \left\{ s_i, \frac{\text{rem}_i^\phi(t) \cdot s_i}{\min \{D_{ij}, t_{\text{next}}\} - t} \right\} \right\}$$  \hspace{1cm} (4.16)$$

The next section gives the pseudo-code of MOTE and finally, we prove in Section 4.6.3.4 that MOTE does not jeopardize the schedulability of the application during its execution.

<table>
<thead>
<tr>
<th>Algorithm 9: Algorithm $\text{computeT}_{\text{next}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong></td>
</tr>
<tr>
<td>- The current time $t$,</td>
</tr>
<tr>
<td>- A real-time tasks $\tau_i$.</td>
</tr>
<tr>
<td><strong>Output:</strong> Instant $t_{\text{next}}$</td>
</tr>
<tr>
<td><strong>begin</strong></td>
</tr>
<tr>
<td>$n_a \leftarrow$ number of active tasks at time $t$ ;</td>
</tr>
<tr>
<td>$L \leftarrow$ set of the next deadline and possible earliest release instants of each task, ordered by non-decreasing occurring time ;</td>
</tr>
<tr>
<td>$t_{\text{next}} \leftarrow t$ ;</td>
</tr>
<tr>
<td>$\Pi \leftarrow m - (n_a - 1)$ ;</td>
</tr>
<tr>
<td><strong>while</strong> $\left( \Pi &gt; 0 \text{ and } L \neq \phi \right)$ <strong>do</strong></td>
</tr>
<tr>
<td>$e \leftarrow L.$top();</td>
</tr>
<tr>
<td>L.pop() ;</td>
</tr>
<tr>
<td>$t_{\text{next}} \leftarrow e.$occurring.time ;</td>
</tr>
<tr>
<td><strong>if</strong> $\left( e.$task $\neq \tau_i \text{ and } (e.$type $== \text{deadline}) \right)$ <strong>then</strong> $\Pi \leftarrow \Pi + 1$ ;</td>
</tr>
<tr>
<td><strong>else if</strong> $\left( e.$type $== \text{release} \right)$ <strong>then</strong> $\Pi \leftarrow \Pi - 1$ ;</td>
</tr>
<tr>
<td><strong>end while</strong></td>
</tr>
<tr>
<td><strong>return</strong> $t_{\text{next}}$ ;</td>
</tr>
<tr>
<td><strong>end</strong></td>
</tr>
</tbody>
</table>

4.6.3.3 Implementation

Recall that at any time $t$, $s_i$ denotes the execution speed of the active job (if any) of task $\tau_i$. This speed $s_i$ is initialized when $\tau_{i,j}$ is released. The initial value of each $s_i$
4.6 Online slack reclaiming algorithms

Algorithm 10: MOTE algorithm

Input: - A set $\tau$ of real-time tasks,
- A platform $\pi$ composed of $m$ DVFS-identical processors,
- A maximal processor speed $s_{\text{max}}$,
- A minimal processor speed $s_{\text{min}}$,
- An initial identical processor speed $s_{\text{ident}}$.

Output: $\phi$

begin
  At job release time (say job $\tau_{i,j}$):
  $s_i \leftarrow s_{\text{ident}}$;
end procedure

At job dispatching time (say job $\tau_{i,j}$ on processor $\pi_k$):
  if ($m \leq n$) then
    $t_{\text{next}} \leftarrow \text{call computeTnext}(t, \tau_i)$ ;
  else
    $t_{\text{next}} \leftarrow \infty$ ;
  end if
  if ($t_{\text{next}} > t$) then
    $s_i \leftarrow \max\left\{s_{\text{min}}, \min\left\{s_{\text{ident}}, \min \{D_{i,j}, t_{\text{next}} - t\}\right\}\right\}$ ;
  end if
  Dispatch $\tau_{i,j}$ to processor $\pi_k$ ;
  Set the operating speed of $\pi_k$ to $s_i$ ;
end procedure

end

can be set to any value within $[s_{\text{min}}, s_{\text{max}}]$, as long as the application has been asserted schedulable while executing the application under these speeds. Here, the initial speed of every job is set to $s_{\text{ident}}$ passed as an argument to the algorithm. We assume that this identical speed $s_{\text{ident}}$ has been determined beforehand, using an approach such as the one proposed in Section 4.5.1 for instance. Then, the decision to reduce (or not) the execution speed of a job is taken only when this job is dispatched to a processor (either upon its release or when it is waiting for an available processor at the head of the ready-Q and a higher-priority job completes). A detailed description of MOTE is given in Algorithm 10. Moreover, Algorithm 9 shows how to compute $t_{\text{next}}$ with a linearithmic (also called quasilinear) worst-case computing complexity $O(n \cdot \log(n))$, where $n$ is the number of tasks in the application. It is worth noting that the speed reduction by MOTE is applied at most once to each job. Indeed, once the speed of a job (say $\tau_{i,j}$) has been
modified by MOTE, we know from the definition of \( t_{\text{next}} \) that it will not be preempted until its completion. Thus, \( \tau_{i,j} \) will not be (re-)stored in the \( \text{ready-Q} \). However, if the execution speed of a job \( \tau_{i,j} \) is initialized but not modified by MOTE upon its release (i.e., the job is not directly dispatched upon its release), then its speed can possibly be reduced by MOTE in the future, when it will be dispatched to any processor.

### 4.6.3.4 Correctness of MOTE

**Lemma 4.9**

Let \( \text{rem}^s_i(t) \) be the remaining worst-case execution time of the last released job of \( \tau_i \) (say \( \tau_{i,j} \)) at time \( t \) when executing at speed \( s \). For any given speed \( s' \), we have:

\[
\text{rem}^{s'}_i(t) = \text{rem}^s_i(t) \cdot \frac{s}{s'}
\]

**Proof**

By definition of \( \text{rem}^s_i(t) \) we know that in the worst-case, \( \tau_{i,j} \) must execute for \( \text{rem}^s_i(t) \) time units at speed \( s \) to complete. During these \( \text{rem}^s_i(t) \) time units, we know by definition of the speed that \( \tau_{i,j} \) completes \( U \) units of execution while running at speed \( s \). On the other hand, \( \tau_{i,j} \) should execute for \( \text{rem}^{s'}_i(t) \) time units at speed \( s' \) to complete this amount \( U \) of execution units. Then we have

\[
U = s \cdot \text{rem}^s_i(t) = s' \cdot \text{rem}^{s'}_i(t)
\]

and thus

\[
\text{rem}^{s'}_i(t) = \text{rem}^s_i(t) \cdot \frac{s}{s'}
\]

**Lemma 4.10**

After a speed reduction by Expression 4.16, the remaining worst-case execution time of the concerned job, say \( \tau_{i,j} \), is
4.6 Online slack reclaiming algorithms

\[
\text{rem}_{i}^{s'}(t) = \min \{ D_{i,j}, t_{\text{next}} \} - t
\]

Proof

The proof is a direct consequence of Lemma 4.7 and Expression 4.16.

Lemma 4.11

Let τ be any set of sporadic and constrained-deadline tasks, let S be any FJP scheduler complying with the specifications given in Section 4.2.3 and let π be any multiprocessor platform composed of \( m \) independent DVFS-identical processors. If \( \tau \) is schedulable by \( S \) on \( \pi \), then the schedulability is not jeopardized while using MOTE algorithm.

Proof

A speed reduction occurs only at job dispatch time (say \( t \)) when \( t_{\text{next}} > t \). Let \( \tau_{i,j} \) be the job that is dispatched to a processor, say \( \pi_{\ell} \), at time \( t \). Two cases can occur:

Case 1: \( \frac{\text{rem}_{i}^{s}(t)s_{i}}{\min \{ D_{i,j}, t_{\text{next}} \} - t} \geq s_{i} \). In this case, MOTE does not modify the speed \( s_{i} \) and the schedulability of the application is therefore preserved.

Case 2: \( \frac{\text{rem}_{i}^{s}(t)s_{i}}{\min \{ D_{i,j}, t_{\text{next}} \} - t} < s_{i} \). Here, the speed \( s_{i} \) is reduced to \( s_{i}' = \frac{\text{rem}_{i}^{s}(t)s_{i}}{\min \{ D_{i,j}, t_{\text{next}} \} - t} \).

We have shown in Lemma 4.10 that, after the speed reduction, the resulting remaining worst-case execution time of \( \tau_{i,j} \) is equal to \( \text{rem}_{i}^{s'}(t) = \min \{ D_{i,j}, t_{\text{next}} \} - t \). Moreover, we showed in Corollary 4.6 that, while ignoring the execution of \( \tau_{i,j} \), \( \pi_{\ell} \) remains available at least until \( t' = \min(D_{i,j}, t_{\text{next}}) \). Consequently, \( \tau_{i,j} \) can be allocated to \( \pi_{\ell} \) and its speed can be reduced so that its worst-case completion time does not exceed \( t' \). The dispatching time of every other job running on the other processors is not modified. As a result, our algorithm avoids scheduling anomalies and the feasibility of the application is not jeopardized.

In short, Lemma 4.11 is a direct consequence of the computation of \( t_{\text{next}} \). Reducing the execution speed of a job implies that this job will execute for longer. However, MOTE decides to reduce the speed of a processor \( \pi_{k} \) at time \( t \) (via a reduction of the execution
CHAPTER 4. EXPLOITING THE DVFS FRAMEWORK

speed of the job, say $\tau_{i,j}$, which is being dispatched to $\pi_k$) only to make it benefit from the future idle time next to the execution of $\tau_{i,j}$. In other words, the computation of $t_{\text{next}}$ ensures that the “extra-time” allocated to the execution of $\tau_{i,j}$ by reducing its speed is a time during which processor $\pi_k$ would have been idle after the execution of $\tau_{i,j}$ if the speed was not modified. And since we proved in Corollary 4.8 that, after the speed modification, $\tau_{i,j}$ will complete at most at time $\min\{d_{i,j}, t_{\text{next}}\}$, the schedulability of the application is not jeopardized.

4.6.4 Combination MORA – MOTE

The two methods MORA and MOTE can be combined in order to further reduce the energy consumption. We call this combination MORAOTE hereafter and this combination is made possible thanks to the following observation.

Using MORA, we proved in Lemma 4.4 that no job completes later in the actual schedule than in the offline schedule. That is, the application meets all the job deadline in the actual schedule as long as all the deadlines are met in the offline schedule.

This is the reason why MORA can also be combined with offline algorithm as mentioned in Section 4.6.2.1. In a more general way, the offline speeds $s_{\text{off}}$ of every job $J_i$ can be set to any available speed as long as all these offline speeds ensure that all the deadlines are met while executing the task set upon the $m$ processors. Consequently (and following the same idea), an elegant way to combine MORA and MOTE consists in applying MOTE to the offline schedule of MORA—That is, MOTE modifies only the offline speed of the jobs $J_i$. We prove in Lemma 4.11 that the offline schedule remains feasible while using MOTE and thus, it holds from the above observation that applying MORA to this “MOTE-offline-schedule” generates in turn a feasible actual schedule. A pseudo-code of this combination is given by Algorithm 11.

312
Algorithm 11: MORAOTE algorithm

Input:
- A set $\tau$ of real-time tasks,
- A platform $\pi$ composed of $m$ DVFS-identical processors,
- A maximal processor speed $s^{max}$,
- A minimal processor speed $s^{min}$.

Output: $\phi$

```
begin
    Initialization step
        Determine an initial identical processor speed $s_{ident}$ ;
        // $s_{ident}$ allows to respect all deadlines and it holds that $s_{min} \leq s_{ident} \leq s^{max}$.
    end procedure

At job release (say $\tau_{ij}$) at time $t$:

begin
    Update the $\alpha$-queue according to $\alpha$-Rule 4.4 ;
    Insert the value of $C_i$ into the $\alpha$-queue according to $\alpha$-Rule 4.3 ;
    $s_i \leftarrow s_{ident}$ ;
    end procedure

When any job $\tau_{ij}$ is dispatched to any processor $\pi_k$ in the offline schedule at time $t$:

begin
    Update the $\alpha$-queue according to $\alpha$-Rule 4.4 ;
    // Here starts the MOTE mechanism
    if ($m \leq n$) then $t_{next} \leftarrow$ call computeNext($t, \tau$) ;
    else $t_{next} \leftarrow \infty$ ;
    if ($t_{next} > t$) then
        $s_i^\text{off} \leftarrow \min\{s_i^\text{off}, \min_{\tau_j, t_{next} \geq t} |s_j^\text{off}|\}$;
        if ($s_i^\text{off} < s^{min}$) then $s_i^\text{off} \leftarrow s^{min}$ ;
        // End of MOTE mechanism
        $s_i \leftarrow \frac{rem(t)}{\sum(\pi)} \cdot s_i^\text{off}$ ;
        if ($a$ job $\tau_{ij} \neq \tau_{ij}$ is running on $\pi_k$) then Preempt $\tau_{ij}$ ;
        Dispatch $\tau_{ij}$ to processor $\pi_k$ ;
        Set the operating speed of $\pi_k$ to $s_i$ ;
    end procedure

When any processor $\pi_k$ is getting idle at time $t$:

begin
    Update the $\alpha$-queue according to $\alpha$-Rule 4.4 ;
    nextdisp$^\text{off} \leftarrow \alpha$-queue.computeNextDisp($\tau_{ij}, t$) ;
    // for each waiting job at time $t$ in the actual schedule
    foreach $\tau_{ij}$ in active$(\tau, t)$ do
        disp$^\text{off} \leftarrow \alpha$-queue.computeDisp($\tau_{ij}, t$) ;
        $L_i \leftarrow \min$[nextdisp$^\text{off}$, disp$^\text{off}$] $-$ $t$ ;
        $s'_{ij} \leftarrow \frac{rem(t)}{\sum(\pi)} \cdot s_i^\text{off}$ ;
        $s''_{ij} \leftarrow \frac{rem(t)}{\sum(\pi)} \cdot s_i^\text{off}$ ;
        $\Delta E_i \leftarrow E_i \left(\frac{rem(0)}{\pi} \cdot s'_{ij}\right) - E_i \left(\frac{rem(0)}{\pi} \cdot s''_{ij}\right)$ ;
        if $\Delta E_i \leq 0 \forall \tau_{ij} \in$ active$(\tau, t)$ then $\tau_{ij} \leftarrow$ job with the highest priority ;
        else $\tau_{ij} \leftarrow$ job with the largest $\Delta E_i$ ;
        if $\tau_{ij} \neq \phi$ then
            $s_i \leftarrow s'_{ij}$ ;
            Dispatch $\tau_{ij}$ to processor $\pi_k$ ;
            Set the operating speed of $\pi_k$ to $s_i$ ;
        else
            Idle the processor $\pi_k$ ;
            Set the operating speed of $\pi_k$ to the minimal available speed $s^{min}$ ;
    end procedure
end
```
4.7 Simulation results

4.7.1 The process to generate our applications

For sake of simplicity, our simulations are performed while considering only periodic and synchronous tasks, rather than sporadic tasks. The reason is that we need to simulate the execution of every generated set of tasks several times in succession (one time for each energy-aware method so that we can compare their efficiency). Using sporadic tasks requires to release jobs randomly in time—as long as for each task $\tau_i$, at least one period $T_i$ has elapsed since its last job release—and thereby, this requires to store all these release times so that the same experiment can be repeated. In order to avoid to store this bountiful amount of data, we have thus opted for periodic and synchronous tasks.

In every simulation, we generated 20,000 task sets denoted by $\tau^1, \tau^2, \ldots, \tau^{20000}$. These task sets are generated by groups of 100 such that the generalized density $\delta_{\text{sum}}(\tau^k)$ of each task set $\tau^k$ of a same group is uniformly selected within $[d, d + 0.05]$. Each group of task sets has its own value of $d$, where $d$ is successively set to 0, 0.05, ..., 9.95, thus leading to 200 groups and then, to a total of 20,000 task sets. Task sets are organized in such groups in order to ensure that the generalized densities are uniformly distributed and cover the whole interval $[0, 10]$. The upper bound on $d$ (i.e., 10) was chosen in order to cover a large number of applications while keeping the simulation time reasonable. Once a generalized density $\delta_{\text{sum}}^k$ is assigned to a task set $\tau^k$, the number and parameters of the tasks are generated according to Algorithm 12. This algorithm generates task sets made up of from 2 to $n_{\text{max}}$ tasks such that

1. the generalized density of the task sets is set to the specified argument $D_{\text{sum}}$ (where $D_{\text{sum}}$ is set to $\delta_{\text{sum}}^k$ for each task set),

2. the individual density of each task does not exceed the specified argument $D_{\text{max}}$.

Note that this algorithm requires 3 arguments: $D_{\text{max}}$, $D_{\text{sum}}$ and the maximum number $n_{\text{max}}$ of tasks. In our simulations, the maximum number $n_{\text{max}}$ of tasks is set to 40; Again, we chose to generate at most 40 tasks in each set in order to keep the simulation time reasonable. Concerning the parameter $D_{\text{max}}$, we will see in the next sections that it has an important impact on the results and we performed simulations for different values of this parameter. The actual execution time of every job $\tau_{i,j}$ is uniformly generated.
4.7 Simulation results

in $[C_i^{10}, C_i]$ in order to reflect the fact that a job may take up to 10 times less than its WCET [31, 62]. Here is a brief description of this application generator:

**line 2–3.** Initially, the set $\tau$ of tasks that will be returned is set to the empty set and the number of tasks is randomly generated within $[[D_{\text{sum}}] + 1, n_{\text{max}}]$, because $n > [D_{\text{sum}}]$ is trivially a necessary condition for having $\delta_i < 1 \forall \tau_i$.

**line 4–9.** Basically, the algorithm generates $n$ densities $\delta_i$ such that $\sum_{i=1}^{n} \delta_i = D_{\text{sum}}$. This is done by dividing a segment of length $D_{\text{sum}}$ into $n$ pieces—the length of each piece can then be used as a density. Hence, this part of the algorithm randomly generates $n - 1$ “cutting points” within $(0, D_{\text{sum}})$.

**line 11.** Once the $n - 1$ cutting points have been ordered by non-decreasing order at line 10, the function “arrangePoints” performs a post-treatment on these points. Basically, this function ensures that the distance between every pair of consecutive points is lower than $D_{\text{max}}$ and larger than a specified value $\epsilon$. This is done so that the tasks densities (that will be ultimately set to these distances) are neither larger than $D_{\text{max}}$ nor equal to an infinitesimal value.

**line 12–20.** The $n$ tasks are generated. For each task $\tau_i$, the algorithm starts by assigning its density to the distance between the two consecutive cutting points points$[i]$ and points$[i + 1]$. Then, it generates the period $T_i$ thanks to function randomPeriod(). We used the algorithm proposed in [52] that assigns periods in such a manner that the hyperperiod$^1$ of the application is limited to reasonably small values (in order to limit the simulation time). At line 15, the deadline $D_i$ is uniformly generated in $[0, T_i]$ and at line 16, the WCET $C_i$ is derived from the density and the deadline. The offset $O_i$ is set to 0 at line 17 since we consider synchronous tasks and at lines 18 and 19, $\tau_i$ is created and added to the task set $\tau$.

Once the 20,000 sets $\tau^k$ of tasks are generated, we associate to each one a number $m^k$ of processors. This number $m^k$ is determined according to the schedulability test proposed in [22], i.e.,

$$m^k \leftarrow \frac{\delta_{\text{sum}}(\tau^k) - \delta_{\text{max}}(\tau^k)}{1 - \delta_{\text{max}}(\tau^k)}$$  \hspace{1cm} (4.17)

and then

$$m^k \leftarrow \begin{cases} 
  n & \text{if } m^k > n \\
  1 & \text{if } m^k < 1 
\end{cases}$$

---

$^1$Recall that the hyperperiod of each task set $\tau^k$ is the least common multiple of the periods of the tasks in $\tau^k$.  

315
CHAPTER 4. EXPLOITING THE DVFS FRAMEWORK

Algorithm 12: Applications generator

Input:
- An upper-bound $D_{\text{sum}}$ on the generalized density,
- An upper-bound $D_{\text{max}}$ on the maximal task density,
- A maximum number of tasks $n_{\text{max}}$.

Output: A set $\tau$ of tasks.

\begin{algorithm}
begin
\begin{algorithmic}
\State $\tau \leftarrow \phi$;
\State $n \leftarrow \text{randomIntegerWithin}([D_{\text{sum}}] + 1, n_{\text{max}}, \text{boundaries}_\text{included})$;
\State points $\leftarrow$ array of $(n + 1)$ floats, indexed from points[1];
\State points[1] $\leftarrow$ 0;
\State points[n + 1] $\leftarrow$ $D_{\text{sum}}$;
\For{$(k = 2 ; k \leq n ; + + k)$}
\State points[k] $\leftarrow$ randomFloatWithin $(0, D_{\text{sum}}, \text{boundaries}_\text{excluded})$;
\EndFor
\State Sort the vector “points” by non-decreasing order of points[i]);
\State call arrangePoints (points, $D_{\text{max}}$);
\For{$(i = 1 ; i \leq n ; + + i)$}
\State $\delta_i \leftarrow$ points[i + 1] – points[i];
\State $T_i \leftarrow$ randomPeriod();
\State $D_i \leftarrow$ randomFloatWithin $(0, T_i, \text{boundaries}_\text{excluded})$;
\State $C_i \leftarrow$ $\delta_i \cdot D_i$;
\State $O_i \leftarrow$ 0;
\State $\tau_i \leftarrow$ createTask $(O_i, C_i, D_i, T_i)$;
\State $\tau \leftarrow$ $\tau \cup \{\tau_i\}$;
\EndFor
\State return $\tau$;
\end{algorithmic}
end
\end{algorithm}

Using this test ensures that each task set $\tau^k$ can be scheduled by $\text{EDF}$ to meet all deadlines upon $m^k$ processors running at maximal speed $s_{\text{max}}$. Then, the execution of each $\tau^k$ is simulated during 100 hyperperiod of $\tau^k$ upon its associated number $m^k$ of processors, using the scheduler $\text{EDF}$. This upper-bound on the simulation time (i.e., 100 hyperperiods) was chosen in order to ensure that every task generates at least 100 jobs. This upper-bound allows to keep the simulation time reasonable and, in our opinion, a minimum of 100 jobs a task is sufficient to draw meaningful statistics. Recall that the ACET of the jobs vary from one job to another and the resulting schedule is therefore different from one hyperperiod to another.
4.7 Simulation results

4.7.2 The simulated methods

The execution of each task set $\tau^k (k = 1, 2, \ldots, 20000)$ is simulated 9 times in a row during 100 hyperperiods, using in turn the 9 energy-aware methods described below.

1. Method $I-EDF_{\text{max}}$ assigns the maximum processor speed $s_{\text{max}}$ to all the processors before scheduling the task set by EDF.

2. Method $I-EDF$ assigns an identical speed to the processors before scheduling the task set by EDF. This identical speed is determined using Algorithm 5 (presented on page 264) and Test 4.6 is used as a reference test in the function sched(EDF, $\tau$, $m$, $s$).

3. Method MORA-EDF schedules the applications by EDF while using MORA. MORA-EDF determines an identical processor speed $s_{\text{ident}}$ using I-EDF and sets the offline speed $s_{\text{off}}^i$ of every task $\tau_i$ to $s_{\text{ident}}$ (at line 1 of Algorithm 8, page 293).

4. Method MOTE-EDF schedules the applications by EDF while using MOTE. This method MOTE-EDF determines an identical processor speed $s_{\text{ident}}$ using I-EDF and, each time a job $\tau_{i,j}$ is released, MOTE-EDF sets its execution speed $s_i$ to $s_{\text{ident}}$ (at line 3 of Algorithm 10, page 309).

5. Method MORAOTE-EDF schedules the application by EDF while using the combined method MORAOTE given in Algorithm 11, page 313. Once again, the identical speed $s_{\text{ident}}$ used at line 10 (and passed as a parameter) is determined using I-EDF.

6. Method $I-EDF^{(k)}$ assigns an identical speed to the processors before scheduling the task set by EDF$^{(k)}$. Both this identical speed and the value of $k$ are obtained by using Algorithm 6 (page 274). During the execution of this algorithm, the speed $s_{\text{ident}}^{\text{edf}}(\tau^{(k)}, m - k + 1)$ (at line 6) is computed using Algorithm 5 (presented on page 264), in which the function sched(EDF, $\tau$, $m$, $s$) is based on Test 4.6.

7. Method MORA-EDF$^{(k)}$, MOTE-EDF$^{(k)}$ and MORAOTE-EDF$^{(k)}$ are defined exactly as MORA-EDF, MOTE-EDF and MORAOTE-EDF (respectively), except that they schedule the applications by EDF$^{(k)}$ and determine the identical speed $s_{\text{ident}}$ using I-EDF$^{(k)}$.

All our simulations were performed while applying the following rule: at any instant in time, if any of the 9 methods above returns a speed $s$ that is not available to
the processors use the available speed $s^+$ which is directly higher than $s$. That is, if one of these methods tries to set the speed of any processor $\pi_k$ to any speed $s \notin \{s^1, s^2, \ldots, s^K\}$ (where $\{s^1, s^2, \ldots, s^K\}$ denotes the set of speeds available to $\pi_k$), then the speed of $\pi_k$ is set to $s^+$ such that

$$s^+ \overset{\text{def}}{=} \min_{i=1}^{K} \{s^i \mid s^i \geq s\}$$

In order to compare these nine methods between them, we need a reference energy consumption for the schedule of each task set $\tau^k$. Ideally the most relevant reference would have been the minimum amount of energy required to schedule $\tau^k$ while meeting every job deadline. However, Yun and Kim proved in [73] that computing the schedule that leads to the minimum amount of energy (sometimes called the “optimal voltage schedule” [32]) is NP-Hard for priority-driven FTP schedulers and uniprocessor platforms. Consequently, it is obvious that the same problem for FJP schedulers (that encompass FTP schedulers) and multiprocessor platforms is also NP-Hard. Thereby, the energy consumption that we use as reference for each task set $\tau^k$ is obtained by the method CLV (for “CLaiRVoYant” method) described below. This method CLV is clairvoyant and unconstrained according to the following interpretations:

- CLV is clairvoyant in the sense that it knows the actual execution time of every job beforehand.
- CLV is unconstrained in the sense that it is not constrained to meet the job deadlines.

Let $H^k$ denote the hyperperiod of $\tau^k$ and let $\tau^k_H$ denote the set of all the jobs executed during the execution of $\tau^k$ during 100 hyperperiods, i.e., in the time interval $[0, 100 \cdot H^k]$. Basically, CLV considers that all the jobs $\tau_{i,j} \in \tau^k_H$ form an unique and fully “parallelisable” job\(^1\) that we denote by $\mathcal{J}^k$; We denote by $W^k$ the processing time of $\mathcal{J}^k$ and we define this processing time by

$$W^k \overset{\text{def}}{=} \sum_{\tau_{i,j} \in \tau^k_H} w_{i,j} \quad (4.18)$$

where $w_{i,j}$ is the actual execution time of each job $\tau_{i,j} \in \tau^k_H$. Based on this unique job $\mathcal{J}^k$, the method CLV works as follows. First, it computes the minimum speed that can execute $\mathcal{J}^k$ within $[0, 100 \cdot H^k]$ assuming that $\mathcal{J}^k$ can be executed in parallel upon the

\(^1\)We mean by “fully parallelisable” the fact that this job executes simultaneously on all the processors during the 100 hyperperiods
4.7 Simulation results

$m^k$ processors. This minimum speed denoted by $s^k_{\text{opt}}$ is computed as follows:

$$s^k_{\text{opt}} \overset{\text{def}}{=} \frac{W^k}{m^k \cdot 100 \cdot H^k} \quad (4.19)$$

According to this definition of $s^k_{\text{opt}}$, it holds that no processor idles in the time interval $[0, 100 \cdot H^k]$ if they run at speed $s^k_{\text{opt}}$ (assuming that all of them execute this fully parallelisable job $J^k$ from time 0 to time $100H^k$). That is, there is no external slack time if $J^k$ is executed simultaneously by the $m^k$ processors in the time interval $[0, 100 \cdot H^k]$. The resulting consumption returned by CLV is then given by

$$E^k_{\text{opt}} \overset{\text{def}}{=} \text{Pwr}_{\text{run}}(s^k_{\text{opt}}) \cdot \sum_{i,j \in T_H} \frac{w_{i,j}}{s^k_{\text{opt}}} \quad (4.20)$$

where $\text{Pwr}_{\text{run}}(s_i)$ denotes the relative power dissipation of the processor while running at speed $s_i$. In the appendix (page 347), we provide the values of $\text{Pwr}_{\text{run}}(s_i)$ (for every available speed $s_i$) for different processor technologies.

In most cases, the speed $s^k_{\text{opt}}$ determined as in Expression 4.19 is not available to the processors. Consequently, if $s^k_{\text{opt}}$ is not available then the function $\text{Pwr}_{\text{run}}(s^k_{\text{opt}})$ used in Expression 4.20 is not defined. The authors of [32] propose a solution to get around this problem. In their Theorem 4.2, they proved that the minimum consumption (if the desired speed $s$ is not available) is obtained by sequentially using the two surrounding available speeds $s^+$ and $s^-$, where $s^+$ is the available speed directly higher than $s$ and $s^-$ is the available speed directly lower than $s$. That is, in the context of our problem, the approach advised in [32] computes the time-instant $t_{\text{switch}}$ such that

$$m \cdot t_{\text{switch}} \cdot s^+ + m \cdot (100H^k - t_{\text{switch}}) \cdot s^- = 100H^k \cdot m \cdot s^k_{\text{opt}}$$

This expression can be interpreted as follows. First, the $m$ processors run at speed $s^+$ in the time interval $[0, t_{\text{switch}}]$. Then at time $t_{\text{switch}}$, the $m$ processors switch their speed to $s^-$ and run at this speed $s^-$ until the time-instant $100H^k$. According to the above equality, $t_{\text{switch}}$ is defined as

$$t_{\text{switch}} \overset{\text{def}}{=} \frac{100H^k \cdot s^k_{\text{opt}} - 100H^k \cdot s^-}{s^+ - s^-}$$
and this approach simply derives the total consumption $E_{\text{tot}}$ from

$$E_{\text{tot}} \overset{\text{def}}{=} m \cdot t_{\text{switch}} \cdot \text{Pwr}_{\text{run}}(s^+) + m \cdot (100H^k - t_{\text{switch}}) \cdot \text{Pwr}_{\text{run}}(s^-)$$

It was proved in [32] that this approach provides the minimum consumption as long as the energy consumption is a convex function of the processor speed. Unfortunately, the energy consumption is not a convex function of the speed (see Figure 4.20 on page 349) for the processor models given in the appendix (page 347). As a consequence, there is no proof that using a combination of the two speeds $s^+$ and $s^-$ surrounding $s_{\text{opt}}^k$ leads to a minimum consumption. For instance, it could be the case that $s_{\text{opt}}^k$ is available to the processor but, because the consumption is not a convex function of the speed, there is a combination of several other available speeds that leads to a lower consumption than $E_{\text{opt}}^k$ defined in Expression 4.20. Also, it could be the case where executing a job at a lower speed consumes more energy than executing the same job at a higher speed and then let the processor idle. For these reasons, our method CLV does not use this approach described in [32]. Rather, CLV redefines Expression 4.20 as follows:

$$E_{\text{opt}}^k \overset{\text{def}}{=} \text{Pwr}_{\text{run}}(s^-) \cdot \sum_{\tau_{i,j} \in T_{\tau_i}^k} \frac{w_{i,j}}{s_{\text{opt}}^k}$$

(4.21)

where $s^-$ is equal to $s_{\text{opt}}^k$ if $s_{\text{opt}}^k$ is available to the processors. Otherwise, $s^-$ is equal to the highest available speed directly lower than $s_{\text{opt}}^k$. The reader should keep in mind that we use CLV not to get a lower-bound on the amount of energy needed to executed $\tau^k$ within $[0, 100 \cdot H^k]$. Rather, we use CLV as a reference to compare the nine methods listed above. The reason is that CLV provides a very low consumption and because of its clairvoyance and unconstrained features, we consider (quite rightly) the consumption of CLV as a suitable target.

4.7.3 Results provided by our offline methods

Our first simulations report only on the results provided by I-EDF$^{\text{max}}$, I-EDF and I-EDF$^{(k)}$. The results from the other methods are presented in the next section. In Figure 4.13, the generalized densities $\delta^k_{\text{sum}}$ (for $k = 1, 2, \ldots, 20000$) of each generated task set $\tau^k$ are ordered in increasing order on the X-axis. The Y-axis gives the consumption of the methods I-EDF$^{\text{max}}$ (in red), I-EDF (in blue) and I-EDF$^{(k)}$ (in green) relative...
to the consumption of CLV (in black) which is considered as 100%. That is, the consumption generated by CLV is considered to be 100% and the consumptions generated by the three methods I-EDF$^{\text{max}}$, I-EDF and I-EDF$^{(k)}$ are normalized with respect to the consumption of CLV. In this simulation, all the generated platforms are composed of only StrongARM SA-1100 processors. The characteristics of this processor model are drawn from [63] and are summarized in Table 4.8 (page 348). We conducted the same simulation with the other processor models presented in Appendix A (page 347) but, because we obtained similar results, we only depict in this chapter the results issued from the use of StrongARM processors. The interested reader may however consult the Appendix B (on page 350) for these additional results.

![Figure 4.13: Comparison between the consumption generated by the three methods I-EDF$^{\text{max}}$ (in red), I-EDF (in blue), I-EDF$^{(k)}$ (in green) and the method CLV (in black), using the consumption model of the processor StrongARM SA-1100. The Y-axis gives the consumption of each method, relative to the consumption of CLV which is considered as 100%. The characteristics of this processor are given in Table 4.8 (page 348).](image)

The simulation whose results are depicted in Figure 4.13 has been carried out while setting the parameters $n_{\text{max}}$ and $D_{\text{max}}$ to 40 and 0.8 (respectively) in the tasks generation process given by Algorithm 12. That is, during the whole simulation process, the num-
CHAPTER 4. EXPLOITING THE DVFS FRAMEWORK

ber of tasks in each task set $\tau^k$ does not exceed 40 while the density of each task does not exceed 0.8. As mentioned earlier, the parameter $D_{\text{max}}$ has a considerable influence on the results provided by our proposed energy-aware methods. Indeed, Figure 4.14 (page 323) depicts the results provided by the same simulation as the previous one, except that the parameter $D_{\text{max}}$ has been set to 0.6 (upper picture) and 0.5 (lower picture). As we can see, the method I-EDF$^{(k)}$ (in green) still provides important energy savings for $D_{\text{max}} = 0.6$ whereas the efficiency of I-EDF (in blue) is decreased (compared to its efficiency for $D_{\text{max}} = 0.8$). Then when $D_{\text{max}} = 0.5$, the efficiency of both I-EDF and I-EDF$^{(k)}$ is much lower than that for $D_{\text{max}} = 0.8$ in Figure 4.13.

In order to measure to what extent the efficiency of I-EDF and I-EDF$^{(k)}$ varies along with the parameter $D_{\text{max}}$, we performed the same simulation as the one described above for every value of $D_{\text{max}}$ within $[0.1, 1]$ by increment of 0.1. The results are depicted on Figure 4.15. More precisely, Figure 4.15a depict the average relative consumption of I-EDF$^{\text{max}}$ (in red), I-EDF (in blue) and I-EDF$^{(k)}$ (in green) for every value of $D_{\text{max}} = 0.1, 0.2, \ldots, 1$. For each method and for each value of $D_{\text{max}}$, the average consumption is given by the height of the corresponding rectangle. On the other hand, Figure 4.15b is less clear but brings more information. For each method and for each value of $D_{\text{max}}$, this figure depicts a box plot that provides (from up to bottom) the maximal recorded consumption, the upper quartile (i.e., the third quartile), the median, the lower quartile (i.e., the first quartile) and the minimal recorded consumption.

4.7.4 Results provided by our online methods

In this section, we analyze the effectiveness of MORA, MOTE and MORAOTE by performing the same simulations as the previous ones, except that we introduce the notion of application-specific parameter $e_i$ of each task $\tau_i$. These application-specific parameters are used to reflect the fact that different tasks may have different instruction sequences and therefore require different function units in the processor, thus leading to different consumption profiles. As mentioned earlier in Section 4.6.2.1 (page 282), these application-specific parameters reflect the difference between the consumption profile of each task and the consumption measured while running benchmarks. In the same section, recall that we denoted by $E_i(R, s)$ the energy consumed by the task $\tau_i$ when executed for $R$ time units at speed $s$. This amount $E_i(R, s)$ of energy was defined as

$$E_i(R, s) \overset{\text{def}}{=} R \cdot (e_i \cdot (Pwr_{\text{run}}(s) - Pwr_{\text{idle}}) + Pwr_{\text{idle}})$$

(4.22)
4.7 Simulation results

Figure 4.14: Comparison between the consumption generated by the three methods \( I\text{-EDF}^{\text{max}} \) (in red), \( I\text{-EDF} \) (in blue), \( I\text{-EDF}^{(k)} \) (in green) and the method CLV (in black), using the consumption model of the processor StrongARM SA-1100. The Y-axis gives the consumption of each method, relative to the consumption of CLV which is considered as 100 %. The parameter \( \mathcal{D}_{\text{max}} \) is set to 0.6 and 0.5 in the upper and lower picture, respectively.
CHAPTER 4. EXPLOITING THE DVFS FRAMEWORK

Average relative consumption of I−EDF−MAX, I−EDF and I−EDF(k) for Dmax in [0.1, 1]

(a) The X-axis ranges the values of $D_{max}$ from 0.1 to 1. The Y-axis displays the relative energy consumption (in %); relative to the consumption of CLV, i.e., the consumption of CLV is 100 %. The height of the rectangles gives the average relative consumption of each method (I-EDF$^{\text{max}}$ in red, I-EDF in blue and I-EDF$^{(k)}$ in green), relatively to the consumption of CLV. For instance, for $D_{max} = 0.1$, the method I-EDF$^{\text{max}}$ consumes approximately 210% × the consumption of CLV (on average).

Relative consumption of I−EDF−MAX, I−EDF and I−EDF(k) for Dmax in [0.1, 1]

(b) The X-axis ranges the values of $D_{max}$ from 0.1 to 1. The Y-axis displays the relative energy consumption (in %); relative to the consumption of CLV, i.e., the consumption of CLV is 100 %. Each box plot provides (from up to bottom) the maximal recorded consumption, the upper quartile, the median, the lower quartile and the minimal recorded consumption.

Figure 4.15: Some statistics about the consumption of I-EDF$^{\text{max}}$ (in red), I-EDF (in blue) and I-EDF$^{(k)}$ (in green). In both figures, the X-axis displays every value of $D_{max}$ while the Y-axis gives the relative consumption of each method (relative to the consumption of CLV).
4.7 Simulation results

where $P_{\text{run}}(s)$ and $P_{\text{idle}}$ are given in Appendix A on page 347 for different processor technologies. In the simulations presented here, the application-specific parameters $e_i$ are uniformly chosen in $[0.8, 1.2]$ so that the consumption of the tasks varies between 80% and 120% of the energy consumption measured while running the benchmarks. 20,000 task sets are randomly generated by using the task generator process introduced in the above Section 4.7.1. The simulated methods are $\text{MORA-EDF}^{(k)}$, $\text{MOTE-EDF}^{(k)}$, $\text{MORAOTE-EDF}^{(k)}$ and CLV introduced in Section 4.7.2. However, we redefine CLV in such a manner that it takes into account the application-specific parameters. That is, the consumption of CLV is now obtained from a combination between Expression 4.21 and Expression 4.22 above, leading to

$$E^{k}_{\text{opt}} \stackrel{\text{def}}{=} \sum_{\tau_i \in T} \left( \frac{w_{i,j}}{s^{k}_{\text{opt}}} \cdot (e_i \cdot (P_{\text{run}}(s^-) - P_{\text{idle}}) + P_{\text{idle}}) \right)$$

where $s^-$ is equal to $s^{k}_{\text{opt}}$ if $s^{k}_{\text{opt}}$ is available to the processors. Otherwise, $s^-$ is equal to the highest available speed directly lower than $s^{k}_{\text{opt}}$.

Figure 4.16 displays the relative consumption of the four methods $l\text{-EDF}^{\text{max}}$ (in red), $\text{MORA-EDF}^{(k)}$ (in light green), $\text{MOTE-EDF}^{(k)}$ (in dark green) and $\text{MORAOTE-EDF}^{(k)}$ (in gold), i.e., relative to the consumption of CLV redefined above. In Figure 4.16a, $D_{\text{max}}$ is set to 0.6 whereas it is set to 0.5 in Figure 4.16b. As we can observe in these figures, the efficiency of the three methods $\text{MORA-EDF}^{(k)}$, $\text{MOTE-EDF}^{(k)}$ and $\text{MORAOTE-EDF}^{(k)}$ is very sensitive to the value of the parameter $D_{\text{max}}$. Similarly to what we presented for the offline methods, Figure 4.17 presents some statistics about each method for every value of $D_{\text{max}}$ in $[0.1, 1]$. For sake of clarity, Figures 4.17a and 4.17b use the following colors: $l\text{-EDF}^{\text{max}}$ in red, $l\text{-EDF}^{(k)}$ in glowing green, $\text{MORA-EDF}^{(k)}$ in white, $\text{MOTE-EDF}^{(k)}$ in gray and $\text{MORAOTE-EDF}^{(k)}$ in gold. The X-axis of both Figures 4.17a and 4.17b ranges the values of $D_{\text{max}}$ from 0.1 to 1. In Figure 4.17a, the height of the rectangles gives the average consumption of each method, relatively to the consumption of CLV.

By examining Figure 4.17b with meticulous care, one can observe two phenomena:

1. The minimum recorded consumption of $\text{MORAOTE-EDF}^{(k)}$ for $D_{\text{max}} = 0.9$ is below 100%, i.e., below the consumption provided by CLV. This asserts the following two statements:
   
   (a) CLV does not provide a lower-bound on the amount of energy needed to
CHAPTER 4. EXPLOITING THE DVFS FRAMEWORK

Figure 4.16: Comparison between the consumption generated by the four methods I-EDF$^{max}$ (in red), MORA-EDF$^{k}$ (in light green), MOTE-EDF$^{k}$ (in dark green), MORAOTE-EDF$^{k}$ (in gold) and the method CLV, using the consumption model of the processor StrongARM SA-1100. The Y-axis gives the consumption of each method, relative to the consumption of CLV which is considered as 100%. The parameter $D_{max}$ is set to 0.6 and 0.5 in the upper and lower picture, respectively.
4.7 Simulation results

(a) The X-axis ranges the values of $D_{\text{max}}$ from 0.1 to 1. The Y-axis displays the relative energy consumption (in %); relative to the consumption of CLV, i.e., the consumption of CLV is 100 %. The height of each rectangle gives the average relative consumption of each method, relatively to the consumption of CLV.

(b) The X-axis ranges the values of $D_{\text{max}}$ from 0.1 to 1. The Y-axis displays the relative energy consumption (in %); relative to the consumption of CLV, i.e., the consumption of CLV is 100 %. Each box plot provides (from up to bottom) the maximal recorded consumption, the upper quartile, the median, the lower quartile and the minimal recorded consumption.

Figure 4.17: Some statistics about the consumption of $I$-EDF$^{\text{max}}$ (in red), $I$-EDF($k$) (in glowing green), MORA-EDF($k$) (in white), MOTE-EDF($k$) (in gray) and MORAOTE-EDF($k$) (in gold). In both figures, the X-axis displays every value of $D_{\text{max}}$ while the Y-axis gives the relative consumption of each method (relative to the consumption of CLV).
execute a given set of tasks, and

(b) in some particular cases, MORAOTE-EDF\((k)\) is very efficient and provides considerable energy savings.

2. From \(D_{\text{max}} = 0.1\) to \(D_{\text{max}} = 1\), the energy reductions generated \textit{exclusively} by MORA-EDF\((k)\) and MOTE-EDF\((k)\) vary \textit{antagonistically}.

The second phenomenon can be explained as follows: a part of the energy savings provided by both MORA-EDF\((k)\) and MOTE-EDF\((k)\) is due to the use of I-EDF\((k)\) for computing the offline speeds. Therefore, if this part of the energy savings (the one provided by I-EDF\((k)\)) is ignored and if we compare only the amounts of energy savings due to MORA-EDF\((k)\) and MOTE-EDF\((k)\) solely, we can see that the efficiency of MOTE-EDF\((k)\) increases along with the parameter \(D_{\text{max}}\), whereas the efficiency of MORA-EDF\((k)\) decreases. These two opposite behaviors clearly appear in Figure 4.18 (page 329), where the consumptions of MORA-EDF\((k)\) and MOTE-EDF\((k)\) are expressed \textit{relatively} to the consumption of I-EDF\((k)\). That is, for every value of parameter \(D_{\text{max}}\) from 0.1 to 1, the consumption of I-EDF\((k)\) is considered as 100\% and the consumption of both MORA-EDF\((k)\) and MOTE-EDF\((k)\) are normalized, i.e., they are expressed as a percentage of the consumption generated by I-EDF\((k)\). As we can observe:

1. MORA-EDF\((k)\) (in white) is very effective when \(D_{\text{max}} = 0.1\) and its efficiency decreases when the value \(D_{\text{max}}\) increases. For \(D_{\text{max}} \in [0.6, 1]\), the energy saving provided by MORA-EDF\((k)\) is negligible.

2. MOTE-EDF\((k)\) (in gray) is not effective for \(D_{\text{max}} \in [0, 0.6]\) and then, its efficiency increases for \(D_{\text{max}} > 0.6\).

3. MORAOTE-EDF\((k)\) (in gold) benefits from the efficiency of both MORA-EDF\((k)\) and MOTE-EDF\((k)\).

The opposite behaviors of MORA-EDF\((k)\) and MOTE-EDF\((k)\) can be explained intuitively, without requiring any formal proofs. The reason resides in Expression 4.17 (page 315). This expression returns the number \(m^k\) of processors used during the simulation of each task set \(\tau^k\) and because of the denominator \((1 - \delta_{\text{max}}(\tau^k))\), the returned number \(m^k\) of processors is highly sensitive to the maximal task density \(\delta_{\text{max}}(\tau^k)\); \(m^k\) is large when \(\delta_{\text{max}}(\tau^k)\) is large. Since tasks are more likely to have a large density when \(D_{\text{max}}\) is large, it holds (by transitivity) that \(m^k\) is large when \(D_{\text{max}}\) is large. This relation is illustrated on Figure 4.19, where we can see that the ratio \(\frac{m}{n}\) grows along with
4.7 Simulation results

parameter $D_{\text{max}}$. Conceptually, MORA saves energy by profiting from the presence of waiting jobs (since the internal slack time is offered to the waiting jobs) whereas MOTE profits from idle processors (and thus from the absence of waiting jobs). As the ratio $\frac{m}{n}$ tends to 1 (i.e., when $D_{\text{max}}$ is high), the jobs are more likely to be directly dispatched to a processor as soon as they are released, i.e., without waiting for a processor becomes available. Therefore, MOTE provides significant energy savings whereas the effectiveness of MORA is almost null. On the other hand, when $\frac{m}{n}$ tends to 0 (i.e., when $D_{\text{max}}$ is low), processors tend to consecutively execute several distinct jobs and the jobs are often waiting. As a result, MORA is often able to reclaim unused time and provides important energy savings whereas the effectiveness of MOTE is negligible.

![Figure 4.18:](image)

**Figure 4.18:** This picture represents the average energy savings generated exclusively by the methods MORA-EDF$(k)$ (in white), MOTE-EDF$(k)$ (in gray) and MORAOTE-EDF$(k)$ (in gold), compared to the average energy savings due to the method I-EDF$(k)$. In this figure, we consider the consumption model of the processor StrongARM SA-1100 whose characteristics are given in Table 4.8 (page 348).

In this work, we did not take into account the time needed by the processors in order to change their speed—we assumed a zero time, meaning that a voltage/speed change can be performed *instantaneously*. Unfortunately, this time is not zero in prac-
Figure 4.19: Evolution of the ratio \( \frac{m}{n} \) (displayed on the Y-axis) for different values of the parameter \( D_{\text{max}} \).

tice. For instance, Pouwelse et al. reported in [59] that Strong ARM SA-1100 processors perform a voltage/speed change in maximum 140 \( \mu s \). About the Crusoe Transmeta TM5400 (for which simulation results are presented in Appendix 4.8), it is reported in [39] that this processor can scale its voltage up or down in less than 20 microseconds. More precisely, the Crusoe Transmeta TM5400 is equipped with the “Longrun Power Management” facility that manages every speed modification. When the software requests for a modification of the speed, Longrun starts scaling the frequency of the chip. When the phase-lock loop (PLL) locks onto the desired clock rate, LongRun modifies the voltage. By always keeping the clock frequency within the limits required by the voltage, LongRun avoids any clock skewing or other undesirable effects. The worst-case scenario of a full swing from 1.1 V to 1.6 V and from 200 to 700 MHz takes only 280 microseconds. However, the processor does not stall during the swing, as a mobile Pentium III does during a SpeedStep adjustment [39]. The processor keeps executing instructions, stalling only when the PLL relocks onto the new frequency. That does not take longer than 20 microseconds in the worst case, and Transmeta’s engineers say they have never observed a relock taking longer than 10 microseconds [39]. This time overhead is not significant for most real-time systems where task timing parameters are on the order of milli-seconds and is therefore ignored in our experiments. If not
negligible and if the maximal number of speed transitions can be bounded during the execution of each job, the “voltage change overheads” can be incorporated into their worst-case execution time (WCET).

4.8 Conclusion

In this chapter, we tackled the energy-aware scheduling problem of real-time tasks upon identical multiprocessor platforms. First, we proposed two offline algorithms; the first one returns an identical speed while the second one assigns an individual speed to each processor. Then, we proposed two online strategies: MORA and MOTE. MORA is a slack reclamation scheme that detects early task completions and slows down the processors accordingly. On the other hand, MOTE anticipates the intervals of time during which the processors will idle and intelligently adapts the current processor speeds. We formally proved that both MORA and MOTE do not jeopardize the schedulability of the application and we showed how these techniques can be combined together. Finally, we performed simulations showing that all these methods can significantly reduce the energy consumption of the processors.

Currently this work addresses the impact of the proposed scheduling algorithms only on the dynamic consumption of the overall microprocessor consumption. Proposed methods do not take into account the power dissipated to hold the circuit state and/or power dissipation due to the imperfections of the physical implementation (leakage power dissipation component). However it is a very well known fact that for integrated circuits manufactured with technologies below 130 nm, and especially with current 90 nm and 65 nm technologies (or lower), the leakage power dissipation component becomes very important and comparable to the dynamic power dissipation [30]. A significant research effort has been provided, and is still deployed on the leakage power dissipation reduction techniques. Proposed methods target not only low-level, hardware actions (such as clock gating) but also higher-level (operating system) actions forcing the processor to enter one of the multiple low-power dissipation modes for better trade-off between power saving and wake-up time (see [29] as an example). The problem of the increased leakage power dissipation of the sub-micron technologies should be the main motivation for future work, in which the existing controllable parameters of our scheduling algorithms (voltage and frequency) should be extended.
with a processor switch-off parameter.

Among the promising paths for future research direction, it could be interesting to specialize MORA so that it takes into account more practical constraints such as preemption costs, migration costs and time overheads due to the multiple voltage and frequency switching. Also, one should investigate “aggressive” slack reclamation schemes in which algorithms anticipate early task completions, based on statistical informations about tasks. This concept has already been exploited by some uniprocessor energy-aware algorithms (see the AGR algorithm proposed in [7] for instance).
References


CHAPTER 4. EXPLOITING THE DVFS FRAMEWORK


REFERENCES


CHAPTER 4. EXPLOITING THE DVFS FRAMEWORK


REFERENCES


[38] Rhan Ha and Jane W.S. Liu. Validating timing constraints in multiprocessor and distributed real-time systems. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1993.


CHAPTER 4. EXPLOITING THE DVFS FRAMEWORK


REFERENCES


Embedded real-time systems are an important field of the computer systems market, which is growing fast and daily. For the design of battery-powered embedded devices, reducing the energy consumption as much as possible has quickly become a key issue, because the performance as well as the number of features of such electronic devices increase much faster than the capacity of the batteries and the demand for energy is therefore getting greater than the offer. A lot of devices in common use are suffering from this excessive demand in energy, such as laptops and mobile phones for instance. In general, most standalone devices that are autonomous in terms of energy are affected by the gap between their energy needs and their available resources. Therefore, various techniques have been developed during the past twenty years so that these devices can match their expected performance while minimizing their consumption. This reduction of their energy consumption also allows them to be equipped with smaller batteries, making themselves smaller and lighter. Informally, power management in embedded systems are desired for many reasons, in particular: prolong the battery life-time, reduce cooling requirements and reduce operating costs for energy and cooling. Furthermore, using less energy reduces potential hazards of overheating, thereby making these devices more reliable and less dangerous [3].

As introduced in Chapter 1, besides guaranteeing reliability and safety, embedded systems are often subject to hard constraints including size and weight limitations, or constraints on thermal dissipation. Among all these embedded systems, real-time systems are subject to timing constraints following which the correctness of an operation depends not only upon its logical correctness, but also upon the time in which the operation is performed. Nowadays, these real-time systems are often built upon multiprocessor platforms because of their high-computational requirements and because multiprocessing often significantly simplifies the design. As pointed out in [1], another advantage is that multiprocessor systems are more energy efficient than equally powerful uniprocessor platforms, because raising the frequency of a single processor results in a multiplicative increase of the energy consumption while adding processors leads to an additive increase. As real-time systems become more complex, they are
often implemented using heterogeneous multiprocessor architectures (i.e. processors do not necessarily have the same architecture) [2] and the energy-aware real-time scheduling problem upon uniform platforms has recently gained in interest in the real-time research domain. As discussed throughout this thesis, reducing the energy consumption is an issue that concerns every component of the embedded systems, from the application to the hardware layer.

Concerning the application layer, the particular multimode application model has been developed as a promising path to increase the accuracy of the schedulability analyses, hence allowing energy-aware techniques to provide better results. In this model, the whole application is divided into several operating modes, where each mode is in charge of its own set of functionalities, with the interpretation that the application can run in only one mode at a time and running in a mode implies that only the tasks of that mode are executed. As a consequence, modeling real-time applications by such multimode models allows to reflect the fact that some tasks are totally independent from each other since they belong to different modes. This precision provided by this model offers a better granularity which increases the accuracy of the schedulability analyses. However, this model came along with the following problematic. During the execution of a multimode real-time application, switching from the current mode to any other mode requires to substitute the current executing tasks with the set of tasks of the destination mode. This substitution introduces a transient stage, where the tasks of the current and destination modes may be scheduled simultaneously, thereby leading to a possible overload that can compromise the respect of some timing constraints. Therefore, ensuring that all the timing requirements are fulfilled requires not only that a schedulability analysis is performed on the tasks of each mode but also that (i) a protocol for transitioning from one mode to another is specified and (ii) a schedulability analysis for each transition is performed. To the best of our knowledge, the transition protocols presented in this thesis are the very first designed for multiprocessor platforms, and especially for uniform platforms. Chapter 2 provides a complete description of our two transition protocols, together with a detailed schedulability analysis.

Afterward, we addressed the problem of reducing the energy consumption while running embedded applications. Our study starts in Chapter 1 with a detailed analysis in which we identified the key actors in the power dissipation of embedded processors. We distinguished between two main components, namely, the leakage and dynamic
Conclusions and perspectives

power dissipation, and we pointed out the fact that leakage consumption is becoming an important factor of the energy consumption of today technology. For this reason, we investigated how to reduce these two components and this led us to two different approaches.

In Chapter 3 we designed a particular hardware architecture named DMP together with appropriate software solutions. This hardware design is structured in two layers, where each layer can be seen as a multiprocessor platform in which all the processors are identical between them but different from the processors of the other layer. That is, one layer is composed of high-power high-performance processors whereas the other one is composed of low-power low-performance processors. We exhibited the multiple advantages of such a hardware design from both software and practical points of view; in particular we showed that this design is compatible with any processor architecture. Then, we showed that our proposed software methods are particularly well adapted for multimode applications. Our algorithms reduce the complex energy-aware real-time scheduling problem to a simple and well-known bin-packing problem for which an abundant literature is available. Finally we showed that these techniques, by taking as starting point some popular optimization heuristics, provides relevant energy saving compared to the energy consumption of a classical multiprocessor platform design.

In Chapter 4, we focused on the dynamic consumption of the processors and we exploited the Dynamic Voltage and Frequency Scaling (DVFS) framework in order to reduce this consumption as much as possible. This framework has been widely studied over the years because of its remarkable efficiency. The main idea behind it is to reduce the dynamic power dissipation of the processors by intelligently managing both the processor supply voltage and frequency. Modifying the voltage and frequency of the processors directly modifies their processing speed and, in the concern of real-time systems, such modifications have to be carried out very carefully since these systems must guarantee a certain temporal behavior, i.e., they must meet all the timing constraints. The DVFS algorithms that we proposed are designed for identical multiprocessor platforms and can therefore be used on each layer of our proposed DMP platform so that its total consumption is further reduced. We propose two different types of DVFS algorithms, namely, offline and online algorithms. Offline algorithms, on the contrary to online algorithms, decide on the computing speed(s) of the processors at system design-time and then, the determined speeds are never modified at run-time. Inversely,
online DVFS algorithms perform “local” adjustments of the processors speed at runtime in such a manner that the resulting speed of each processor matches its current workload as closely as possible. We demonstrated that all our DVFS algorithms do not jeopardize the schedulability of the applications and we showed how these techniques can be combined together to further improve the energy consumption. Finally, we performed simulations showing that all these methods can significantly reduce the energy consumption of the processors.

Among the promising paths for future research directions, one can cite the studies about the system-wide energy optimization problem. The goal of these studies is to reduce the energy consumption of the whole system rather than focusing only on the consumption of the processors. Recent studies such as [4] for instance propose realistic DVFS energy models that considers CPU, system bus, memory and task set characteristics at multiple frequency settings. In our opinion, the embedded systems are becoming so complex and specialized that the software and hardware can no longer be seen as two independent components of the systems. Rather, we clearly encourage the co-design philosophy following which software programmers and electronic engineers work all together in a same team. During my four years as a PhD-student, I had the privilege of working in two successful departments at Université Libre de Bruxelles, namely, the Department of Computer Science and the Embedded Electronics Group of BEAMS Department. The Department of Computer Science is much more concerned with theoretical problems whereas the Embedded Electronics Group of BEAMS Department focuses on hardware designs. Clearly, all our interactions were beneficial to our two teams and led us to exciting, innovating and successful researches.
Conclusions and perspectives

References


Appendix

A. Processors characteristics

<table>
<thead>
<tr>
<th></th>
<th>Frequency (Mhz)</th>
<th>Corresponding speed</th>
<th>Voltage (V)</th>
<th>Relative Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel XScale</td>
<td>1000</td>
<td>1.0</td>
<td>1.8</td>
<td>100 %</td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>0.8</td>
<td>1.6</td>
<td>56.3 %</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>0.6</td>
<td>1.3</td>
<td>25 %</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>0.4</td>
<td>1.0</td>
<td>10.6 %</td>
</tr>
<tr>
<td></td>
<td>150</td>
<td>0.15</td>
<td>0.75</td>
<td>5 %</td>
</tr>
</tbody>
</table>

Relative power in idle mode: 2.5 %

Table 4.1: Characteristics of the Intel XScale processor family. In [1], the power characteristics are (in mW) 80, 170, 400, 900 and 1600 at frequency 150, 400, 600, 800 and 1000, respectively. The idle power is 40 mW.

<table>
<thead>
<tr>
<th></th>
<th>Frequency (Mhz)</th>
<th>Corresponding speed</th>
<th>Voltage (V)</th>
<th>Relative Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerPC-405LP</td>
<td>333</td>
<td>1.0</td>
<td>1.9</td>
<td>100 %</td>
</tr>
<tr>
<td></td>
<td>266</td>
<td>0.8</td>
<td>1.8</td>
<td>80 %</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.3</td>
<td>1.0</td>
<td>9.6 %</td>
</tr>
<tr>
<td></td>
<td>33</td>
<td>0.1</td>
<td>1.0</td>
<td>2.5 %</td>
</tr>
</tbody>
</table>

Relative power in idle mode: 1.6 %

Table 4.2: Characteristics of the processor PowerPC-405LP. In [4], the power characteristics are (in mW) 19, 72, 600 and 750 at frequency 33, 100, 266 and 333, respectively. The idle power is 12 mW.
### Appendix

<table>
<thead>
<tr>
<th>Frequency (Mhz)</th>
<th>Corresponding speed</th>
<th>Voltage (V)</th>
<th>Relative Power (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>700</td>
<td>1</td>
<td>1.65</td>
<td>100</td>
</tr>
<tr>
<td>600</td>
<td>0.857</td>
<td>1.6</td>
<td>80.59</td>
</tr>
<tr>
<td>500</td>
<td>0.714</td>
<td>1.5</td>
<td>59.03</td>
</tr>
<tr>
<td>400</td>
<td>0.571</td>
<td>1.4</td>
<td>41.14</td>
</tr>
<tr>
<td>300</td>
<td>0.429</td>
<td>1.25</td>
<td>24.6</td>
</tr>
<tr>
<td>200</td>
<td>0.286</td>
<td>1.1</td>
<td>12.7</td>
</tr>
</tbody>
</table>

Relative power in idle mode: unknown

Table 4.3: Characteristics of the processor Transmeta Crusoe TM5400 [3].

<table>
<thead>
<tr>
<th>Frequency (Mhz)</th>
<th>Corresponding speed</th>
<th>Voltage (V)</th>
<th>Relative Power (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>206</td>
<td>1</td>
<td>1.5</td>
<td>100</td>
</tr>
<tr>
<td>195</td>
<td>0.947</td>
<td>1.42</td>
<td>78.9</td>
</tr>
<tr>
<td>180</td>
<td>0.874</td>
<td>1.3</td>
<td>63.2</td>
</tr>
<tr>
<td>165</td>
<td>0.801</td>
<td>1.2</td>
<td>50</td>
</tr>
<tr>
<td>150</td>
<td>0.728</td>
<td>1.15</td>
<td>39.9</td>
</tr>
<tr>
<td>135</td>
<td>0.655</td>
<td>1.1</td>
<td>33.6</td>
</tr>
<tr>
<td>120</td>
<td>0.583</td>
<td>1.08</td>
<td>33</td>
</tr>
<tr>
<td>105</td>
<td>0.51</td>
<td>0.95</td>
<td>19.8</td>
</tr>
<tr>
<td>90</td>
<td>0.437</td>
<td>0.9</td>
<td>15</td>
</tr>
<tr>
<td>75</td>
<td>0.364</td>
<td>0.82</td>
<td>11.8</td>
</tr>
<tr>
<td>60</td>
<td>0.291</td>
<td>0.8</td>
<td>9.44</td>
</tr>
</tbody>
</table>

Relative power in idle mode: unknown

Table 4.4: Characteristics of the processor StrongARM SA-1100 [5].
### Appendix

<table>
<thead>
<tr>
<th>Frequency (Mhz)</th>
<th>Corresponding speed</th>
<th>Voltage (V)</th>
<th>Relative Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>624</td>
<td>1.0</td>
<td>1.55</td>
<td>100 %</td>
</tr>
<tr>
<td>520</td>
<td>0.83</td>
<td>1.45</td>
<td>80.76 %</td>
</tr>
<tr>
<td>416</td>
<td>0.67</td>
<td>1.35</td>
<td>61.62 %</td>
</tr>
<tr>
<td>312</td>
<td>0.5</td>
<td>1.1</td>
<td>40.54 %</td>
</tr>
<tr>
<td>208</td>
<td>0.33</td>
<td>1.15</td>
<td>30.16 %</td>
</tr>
<tr>
<td>104</td>
<td>0.17</td>
<td>0.9</td>
<td>12.54 %</td>
</tr>
<tr>
<td>13</td>
<td>0.02</td>
<td>0.85</td>
<td>4.78 %</td>
</tr>
</tbody>
</table>

Relative power in idle mode: 0.92 %

**Table 4.5:** Characteristics of the processor Intel PXA270. From Table 5-7 in [2], the power characteristics are (in mW) 44.2, 116, 279, 375, 570, 747 and 925 at frequency 13, 104, 208, 312, 416, 520 and 624 respectively. The idle power is 8.5 mW.

**Figure 4.20:** Comparison of the consumption profiles of each processor model presented in this Appendix.
B. Additional experiments for our DVFS methods

In Chapter 4, we reported on some simulation results in Section 4.7. All these simulations were conducted on processors StrongARM SA-1100 and it was claimed that other processor models lead to similar results. In order to convince the reader, we present here the results obtained from similar simulations but considering the other processor models given in Appendix A. This section is structured as follows:

- Figures 4.22 and 4.23 (on pages 371 and 372) show the results obtained for our offline and online methods (respectively) on Transmeta Crusoe TM5400 processors.
- Figures 4.24 and 4.25 (on pages 373 and 374) show the results obtained for our offline and online methods (respectively) on PowerPC-405LP processors.
- Figures 4.26 and 4.27 (on pages 375 and 376) show the results obtained for our offline and online methods (respectively) on Intel PXA270 processors.
- Figures 4.28 and 4.29 (on pages 377 and 378) show the results obtained for our offline and online methods (respectively) on Intel XScale processors.
Appendix

C. Missing proofs of Section 2.8.6

As in the previous section and for the same reasons, we use in the following two proofs the notations idle\(_i\) and comp\(_i\) instead of idle\(_i\)(\(J_i, \pi, \mathcal{P}\)), comp\(_i\)(\(J_i, \pi, \mathcal{P}\)) and comp\(_i\)\(_(3)\)(\(J_i, \pi, \mathcal{P}\)), respectively.

**Lemma 4.12**

If \(J_i\) is ordered by decreasing order of job priority, i.e., \(J_i > J_{i+1}\) \(\forall i\), then it holds \(\forall i \in [1, n]\) that

\[
\sum_{j=2}^{m} (idle_j^i - idle_{j-1}^i) \cdot s_{j-1} \geq \left( idle_m^i - \frac{\sum_{j=1}^{i-1} c_j}{s(1)} \right) \cdot s_m \cdot \frac{s_x}{\sum_{j=1}^{s} s_j} \quad (4.23)
\]

where \(s_x\) denotes any critical speed of \(\pi\).

**Proof**

We define the *equilibrium* instant equ\(_i\) as the instant in the schedule of the jobs with a higher priority than \(J_i\) (i.e., the job \(J_1, J_2, \ldots, J_{i-1}\)) such that

\[
\sum_{k | idle_k^i > equ_i} \left( idle_k^i - equ_i \right) \cdot s_k = \sum_{k | idle_k^i < equ_i} \left( equ_i - idle_k^i \right) \cdot s_k \quad (4.24)
\]

Figure 4.21a provides a visualization of the equilibrium instant equ\(_i\). Informally, this instant equ\(_i\) is the only instant in the schedule of the jobs \(J_1, J_2, \ldots, J_{i-1}\) such that the amounts of execution units executed in the yellow and red areas are equivalent. As showed below, the equilibrium instants equ\(_i\) (for \(i = 1, \ldots, n\)) are given by

\[
equ_i = \frac{\sum_{j=1}^{i-1} c_j}{s(1)}
\]

Indeed, Equality 4.24 yields
$$\sum_{k \mid \text{idle}_{i}^j \geq \text{equ}} \left( \text{idle}_{i}^j - \text{equ}^j \right) \cdot s_k - \sum_{k \mid \text{idle}_{i}^j < \text{equ}} \left( \text{equ}^j - \text{idle}_{i}^j \right) \cdot s_k = 0$$

$$\iff \sum_{i=1}^{m} \left( \text{idle}_{i}^j - \text{equ}^j \right) \cdot s_k = 0$$

$$\iff \sum_{k=1}^{m} \text{idle}_{i}^j \cdot s_k - \sum_{k=1}^{m} \text{equ}^j \cdot s_k = 0$$

$$\iff \sum_{j=1}^{i} c_j - \sum_{k=1}^{m} \text{equ}^j \cdot s_k = 0$$

$$\iff -\sum_{k=1}^{m} \text{equ}^j \cdot s_k = -\sum_{j=1}^{i-1} c_j$$

$$\iff \text{equ}^i = \sum_{j=1}^{i} c_j \frac{1}{s(1)}$$

Let $\gamma \overset{\text{def}}{=} \left\{ \pi_1, \pi_2, \ldots, \pi_r \right\}$ be the subset of processors upon which $J_i$ may be partially (or totally) executed before time $\text{equ}^i$, i.e., $\pi_k \in \gamma$ if and only if $\text{idle}_{i}^k < \text{equ}^i$. In Figure 4.21a, we have $\gamma \overset{\text{def}}{=} \left\{ \pi_1, \pi_2, \ldots, \pi_p \right\}$ (where $r = p$) and obviously, it results from the definition of $\gamma$ that $r \leq m - 1$ since $\text{idle}_{i}^m \geq \text{equ}^i \forall i$. Let $\omega_k$, $1 \leq k \leq r$ denote the amount of execution units that $J_i$ may execute on every processor $\pi_k \in \gamma$ before time $\text{equ}^i$. That is, $\forall k \in [1, r]$

$$\omega_k \overset{\text{def}}{=} \begin{cases} \left( \text{idle}_{i}^k - \text{idle}_{i}^{k+1} \right) \cdot s_k & \text{if } k < r \\ \left( \text{equ}^i - \text{idle}_{i}^k \right) \cdot s_k & \text{otherwise (} k = r \right)$$

From the definition of $\omega_k$, $1 \leq k \leq r$, it holds that

$$\sum_{j=2}^{m} \left( \text{idle}_{i}^j - \text{idle}_{i}^{j-1} \right) \cdot s_{j-1} \geq \sum_{k=1}^{r} \omega_k$$

and the remainder of the proof consists in determining a lower-bound on $\sum_{k=1}^{r} \omega_k$. To do so, we divide the proof into two parts. In the first one, we determine a lower-bound on the amount of execution units that can be executed in the entire yellow area. Then in the second part, we determine the shape of the yellow area that minimizes the amount of execution units that $J_i$ can execute in it, i.e., the shape that minimizes $\sum_{k=1}^{r} \omega_k$.

**Part 1.** According to the definition of the equilibrium instant $\text{equ}^i$, the amount $E_{\text{yellow}}$ of execution units that can be executed in the entire yellow area is equivalent to the amount $E_{\text{red}}$ of execution units that can be executed in the red area.
Therefore, determining a lower-bound on $E_{\text{yellow}}$ is equivalent to determine a lower-bound on $E_{\text{red}}$. From the definition of $\text{equ}^i$, we know that exactly $(\text{idle}_m^i - \text{equ}^i) \cdot s_m$ execution units are executed on processor $\pi_m$ in the time interval $[\text{equ}^i, \text{idle}_m^i]$. And from the definition of $\gamma$, we know that processor $\pi_m \not\in \gamma$ and thus, $\pi_m$ never belongs to the yellow area. As a result, at least $(\text{idle}_m^i - \text{equ}^i) \cdot s_m$ execution units are executed in the red area and we get $E_{\text{red}} \geq (\text{idle}_m^i - \text{equ}^i) \cdot s_m$, leading to

$$E_{\text{yellow}} \geq (\text{idle}_m^i - \text{equ}^i) \cdot s_m$$

**Part 2.** In this second part, we show that the minimum amount of execution units that $J_i$ can achieve in the yellow area is given by $\bar{c}_i$ where

$$\bar{c}_i = (\text{idle}_m^i - \text{equ}^i) \cdot s_m \cdot \frac{s_x}{\sum_{j=1}^{m-1} s_j}$$

and $s_x$ is any critical speed of $\pi$. This sub-proof is obtained by reducing the problem to a problem of geometry. First, notice that the shape of the yellow area can only be a staircase composed of at most $(m - 1)$ steps. Indeed, we already mentioned above that processor $\pi_m$ cannot belong to the yellow area since it cannot belong to the set $\gamma$ defined previously. That is, any acceptable shape of the yellow area can be divided into at most $(m - 1)$ rectangles $R_1, R_2, \ldots, R_{m-1}$ as illustrated in Figure 4.21b. Let us introduce the following notations, for which a graphical interpretation is also depicted in Figure 4.21b.

- **width($R_j$)** denotes the width of the rectangle $R_j$. Notice that in the context of our problem, the width of any rectangle $R_j$ is an interval of time.

- **height($R_j$)** denotes the height of the rectangle $R_j$. As mentioned above, the height of any rectangle corresponds to a certain processor index and must therefore lies within $[1, m - 1]$. Moreover, in the context of our problem, the height of every rectangle $R_j$ must be higher than that of the rectangle $R_{j-1}$.

- **area($R_j$)** denotes the amount of execution units that can be executed in $R_j$. Hereafter, we name this quantity the **area** of $R_j$ and we define it as

$$\text{area}(R_j) \overset{\text{def}}{=} \text{width}(R_j) \cdot \sum_{k=1}^{\text{height}(R_j)} s_k$$
Appendix

- $\text{avbl}(R_j)$ denotes the amount of execution unit that job $J_i$ can execute in the rectangle $R_j$, i.e.,

$$
\text{avbl}(R_j) \overset{\text{def}}{=} \text{width}(R_j) \cdot \text{height}(R_j)
$$

Notice that the notation $\text{avbl}(R_j)$ stands for “available($R_j$)”.

Now, let us focus on any rectangle $R_j$ of the shape such that $\text{height}(R_j) = \ell \neq x$ (where $s_x$ denotes any critical speed of $\pi$). It holds for $R_j$ that

$$
\text{width}(R_j) = \frac{\text{area}(R_j)}{\sum_{k=1}^{x} s_k}
$$

and

$$
\text{avbl}(R_j) = \text{width}(R_j) \cdot s_{\ell} = \text{area}(R_j) \cdot \frac{s_{\ell}}{\sum_{k=1}^{x} s_k}
$$

If we transform the rectangle $R_j$ into the rectangle $R'_j$ by changing its height to $x$ while keeping its area constant, it yields

$$
\text{height}(R'_j) = x
$$

$$
\text{area}(R'_j) = \text{area}(R_j)
$$

$$
\text{width}(R'_j) = \frac{\text{area}(R'_j)}{\sum_{k=1}^{x} s_x} = \frac{\text{area}(R_j)}{\sum_{k=1}^{x} s_x}
$$

$$
\text{avbl}(R'_j) = \text{width}(R'_j) \cdot s_x = \text{area}(R_j) \cdot \frac{s_x}{\sum_{k=1}^{x} s_x}
$$

Thus, we have

$$
\text{avbl}(R_j) = \text{area}(R_j) \cdot \frac{s_{\ell}}{\sum_{k=1}^{x} s_{\ell}}
$$

and

$$
\text{avbl}(R'_j) = \text{area}(R_j) \cdot \frac{s_x}{\sum_{k=1}^{x} s_x}
$$
and since by definition of a critical speed, \( s_x \) is such that \( \frac{s_x}{\sum_{j=1}^{m} s_j} = \min_{i=1}^{m} \left\{ \frac{s_i}{\sum_{j=1}^{m} s_j} \right\} \), it holds that

\[
\text{avbl}(R_j) \geq \text{avbl}(R'_j)
\]

As a result, we can minimize the value \( \text{avbl}(R_j) \) of every rectangle \( R_j \) of the shape by performing the same transformation as the one presented above, i.e., by setting its height to \( x \) without modifying its area (only its width varies). Ultimately, all these transformations lead to a shape composed of a single rectangle \( R \) of height \( x \).

Assuming that the area of \( R \) is the lower-bound \( (\text{idle}^i_m - \text{equ}^i) \cdot s_m \) determined in Part 1, the minimum amount of execution time that \( J_i \) can achieve in the yellow area is given by \( \tilde{c}_i \), where

\[
\tilde{c}_i = \text{avbl}(R) = \text{area}(R) \cdot \frac{s_x}{\sum_{k=1}^{i} s_k} = (\text{idle}^i_m - \text{equ}^i) \cdot s_m \cdot \frac{s_x}{\sum_{j=1}^{i} s_j}
\]

In summary, \( \tilde{c}_i \) is a lower-bound on \( \sum_{k=1}^{i} w_k \) and \( \sum_{k=1}^{i} w_k \) is a lower-bound on \( \sum_{j=2}^{i} (\text{idle}^j_m - \text{idle}^j_{m-1}) \cdot s_{j-1} \). Therefore, it holds that

\[
\sum_{j=2}^{i} (\text{idle}^j_m - \text{idle}^j_{m-1}) \cdot s_{j-1} \geq \tilde{c}_i \geq (\text{idle}^i_m - \text{equ}^i) \cdot s_m \cdot \frac{s_x}{\sum_{j=1}^{i} s_j} \geq \left( \text{idle}^i_m - \frac{\sum_{j=1}^{i-1} c_j}{s(1)} \right) \cdot s_m \cdot \frac{s_x}{\sum_{j=1}^{i-1} s_j}
\]

The lemma follows.

**Lemma 4.13**

If \( J \) is ordered by decreasing order of \( P \)-priorities, i.e., \( J_i > J_{i+1} \forall i \), then an upper-bound on the completion time of job \( J_i \) is given by \( \text{comp}^{(3)}_i(J, \pi, P) \), where (using our simplified notations)
Appendix

\[
\text{comp}_i^{(3)} \overset{\text{def}}{=} \left(1 - \frac{s_x}{\sum_{j=1}^{x} s_j}\right) \cdot \text{idle}_m^i + \left(\frac{c_i}{s_m} + \frac{\sum_{j=1}^{i-1} c_j}{s(1) \cdot \sum_{j=1}^{i-1} s_j}\right)
\] (4.25)

Proof

The proof directly follows from Lemmas 2.16 and 4.12. Indeed, we know from Lemma 2.16 that an upper-bound on the completion time of every job $J_i$ is given by

\[
\text{idle}_m^i = \frac{\sum_{j=2}^{m} (\text{idle}_j^i - \text{idle}_{j-1}^i) \cdot s_{j-1}}{s_m} + \frac{c_i}{s_m}
\]

and from Lemma 4.12 we have

\[
\sum_{j=2}^{m} (\text{idle}_j^i - \text{idle}_{j-1}^i) \cdot s_{j-1} \geq \left(\text{idle}_m^i - \frac{\sum_{j=1}^{i-1} c_j}{s(1)}\right) \cdot s_m \cdot \frac{s_x}{\sum_{j=1}^{x} s_j}
\]

Thus, it holds that

\[
\text{idle}_m^i = \frac{\sum_{j=2}^{m} (\text{idle}_j^i - \text{idle}_{j-1}^i) \cdot s_{j-1}}{s_m} + \frac{c_i}{s_m} \leq \text{idle}_m^i - \frac{\left(\text{idle}_m^i - \frac{\sum_{j=1}^{i-1} c_j}{s(1)}\right) \cdot s_m \cdot \frac{s_x}{\sum_{j=1}^{x} s_j} + \frac{c_i}{s_m}}{s_m}
\]

\[
\leq \left(1 - \frac{s_x}{\sum_{j=1}^{x} s_j}\right) \cdot \text{idle}_m^i + \left(\frac{c_i}{s_m} + \frac{\sum_{j=1}^{i-1} c_j}{s(1) \cdot \sum_{j=1}^{i-1} s_j}\right)
\]

and the instant

\[
\text{comp}_i^{(3)} \overset{\text{def}}{=} \left(1 - \frac{s_x}{\sum_{j=1}^{x} s_j}\right) \cdot \text{idle}_m^i + \left(\frac{c_i}{s_m} + \frac{\sum_{j=1}^{i-1} c_j}{s(1) \cdot \sum_{j=1}^{i-1} s_j}\right)
\]

is therefore also an upper-bound on the completion time of $J_i$.

Lemma 4.14

Let $\mathcal{P}$ denote any job priority assignment. Suppose that $J$ is ordered by decreasing $\mathcal{P}$-priorities, i.e., $J_i >_{\mathcal{P}} J_{i+1}$ $\forall i$, and suppose that $J$ is scheduled by $\mathcal{P}$ upon $\pi$. If $\exists \ell$ such that
 Appendix

\[ J_\ell \] is not the lowest priority job according to \( \mathcal{P} \), i.e., \( \ell < n \)

and \( \text{comp}_{\ell}^{(3)}(J, \pi, \mathcal{P}) = \max_{i=1}^{n} \{ \text{comp}_{i}^{(3)}(J, \pi, \mathcal{P}) \} \)

then there exists a job priority assignment \( \mathcal{P}' \) such that

\[ J_\ell \] is the lowest priority job according to \( \mathcal{P}' \)

and \( \text{comp}_{\ell}^{(3)}(J, \pi, \mathcal{P}) = \max_{i=1}^{n} \{ \text{comp}_{i}^{(3)}(J, \pi, \mathcal{P}) \} \) (4.26)

and \( \text{comp}_{\ell}^{(3)}(J, \pi, \mathcal{P}) \geq \text{comp}_{\ell}^{(3)}(J, \pi, \mathcal{P}) \) (4.27)

Proof

Let \( \mathcal{P}' \) be job priority assignment derived from \( \mathcal{P} \) as follows:

\[ J_1 >_{\mathcal{P}'} J_2 >_{\mathcal{P}'} \cdots >_{\mathcal{P}'} J_{\ell-1} >_{\mathcal{P}'} J_{\ell+1} >_{\mathcal{P}'} \cdots >_{\mathcal{P}'} J_n >_{\mathcal{P}'} J_\ell \]

That is, \( \mathcal{P}' \) is identical to \( \mathcal{P} \) except that \( \mathcal{P}' \) assigns the lowest priority to \( J_\ell \). By hypothesis, we know that \( \text{comp}_{\ell}^{(3)}(J, \pi, \mathcal{P}) = \max_{i=1}^{n} \{ \text{comp}_{i}^{(3)}(J, \pi, \mathcal{P}) \} \) where

\[
\text{comp}_{\ell}^{(3)}(J, \pi, \mathcal{P}) = \left(1 - \frac{s_x}{\sum_{j=1}^{s_x} s_j}\right) \cdot \text{idle}_m(J, \pi, \mathcal{P}) + \left(\frac{c_{\ell}}{s_m} + s_x \cdot \frac{\sum_{j>_{\mathcal{P}'} J_\ell} c_j}{s(1) \cdot \sum_{j=1}^{s_x} s_j}\right)
\]

according to Expression 4.25. Therefore, assigning the lowest priority to \( J_\ell \) as done by \( \mathcal{P}' \) increases the number of jobs with a higher priority than \( J_\ell \), hence increasing both the value of \( \text{idle}_m(J, \pi, \mathcal{P}) \) and the value of \( \sum_{j>_{\mathcal{P}'} J_\ell} c_j \). Consequently, it holds that

\[
\text{comp}_{\ell}^{(3)}(J, \pi, \mathcal{P}) \geq \text{comp}_{\ell}^{(3)}(J, \pi, \mathcal{P})
\]

thus providing Expression 4.27. Now, we can easily prove that Expression 4.26 also holds. Indeed, since \( \mathcal{P} \) and \( \mathcal{P}' \) assign the same priorities to the jobs \( J_1, J_2, \ldots, J_{\ell-1} \), it follows from Expression 4.28 that \( \forall r \in [1, \ldots, \ell - 1] \),

\[
\text{comp}_{r}^{(3)}(J, \pi, \mathcal{P}) = \text{comp}_{r}^{(3)}(J, \pi, \mathcal{P})
\]

and \( \forall r \in [\ell + 1, \ldots, n] \),

\[
\text{comp}_{r}^{(3)}(J, \pi, \mathcal{P}) < \text{comp}_{r}^{(3)}(J, \pi, \mathcal{P})
\]
Appendix

(because job $J_\ell$ has the lowest priority according to $P'$ and thus, the expression of $\text{comp}_r(3)(J, \pi, P')$ considers one job less than the expression of $\text{comp}_r(3)(J, \pi, P)$). In conclusion, it holds that

$$\text{comp}_r(3)(J, \pi, P') \leq \text{comp}_r(3)(J, \pi, P) \quad \forall J_r \neq J_\ell \quad \text{(from Expressions 4.30 and 4.31)}$$

and

$$\text{comp}_r(3)(J, \pi, P) \leq \text{comp}_r(3)(J, \pi, P') \quad \forall J_r \neq J_\ell \quad \text{(by hypothesis)}$$

and

$$\text{comp}_r(3)(J, \pi, P) \leq \text{comp}_r(3)(J, \pi, P') \quad \text{(from Expression 4.29)}$$

By transitivity, this yields $\text{comp}_r(3)(J, \pi, P') \leq \text{comp}_r(3)(J, \pi, P') \quad \forall J_r \neq J_\ell$ and Expression 4.26 follows.

Corollary 4.3

There always exists at least one job priority assignment $P$ such that $J_{\text{low}}$ is the lowest priority job according to $P$ and for all job priority assignments $X$:

$$\text{comp}_{\text{low}}(3)(J, \pi, P) = \max_{i=1}^{n} \{\text{comp}_i(3)(J, \pi, X)\}$$

i.e.,

$$\text{comp}_{\text{low}}(3)(J, \pi, P) \geq \text{maximum makespan}$$

Proof

The proof is a direct consequence of Lemmas 4.14 and 4.13. Indeed, let $P'$ denote any job priority assignment for which $\exists \ell$ such that

$J_\ell$ is not the lowest priority job according to $P'$

and $\forall X$:

$$\text{comp}_r(3)(J, \pi, P') \geq \max_{i=1}^{n} \{\text{comp}_i(3)(J, \pi, X)\}$$

If we define the priority assignment $P$ as the same priority assignment as $P'$ except that $P$ assigns the lowest priority to $J_\ell$, we know from Lemma 4.14 that

$$\text{comp}_r(3)(J, \pi, P) = \max_{i=1}^{n} \{\text{comp}_i(3)(J, \pi, P)\} \quad \text{(4.32)}$$

and

$$\text{comp}_r(3)(J, \pi, P) \geq \text{comp}_r(3)(J, \pi, P') \quad \text{(4.33)}$$
and thus,

\[
\overline{\text{comp}}_\ell^{(3)}(J, \pi, \mathcal{P}) \geq \text{maximum makespan}
\]

Lemma 4.15

There always exists at least one job priority assignment \(\mathcal{P}^{\text{max}}\) such that:

- \(J_{\text{max}}\) is the lowest priority job according to \(\mathcal{P}^{\text{max}}\)
- \(c_{\text{max}} = \max_{j=1}^{n} \{c_j\}\)
- \(\overline{\text{comp}}_\ell^{(3)}(J, \pi, \mathcal{P}^{\text{max}}) \geq \text{maximum makespan}\)

Proof

The proof is made by contradiction. Suppose that there does not exist such a job priority assignment \(\mathcal{P}^{\text{max}}\). Let \(\mathcal{P}^{\text{other}}\) be any job priority assignment such that

- \(J_{\text{low}}\) is the lowest priority job,
- \(\overline{\text{comp}}_\ell^{(3)}(J, \pi, \mathcal{P}^{\text{other}}) \geq \text{maximum makespan}\) (we know from Corollary 4.3 that such a priority assignment \(\mathcal{P}^{\text{other}}\) exists), and
- \(J_{\text{low}}\) is not the (or any) job with the largest processing time.

We show in the following that it is always possible to derive \(\mathcal{P}^{\text{max}}\) from \(\mathcal{P}^{\text{other}}\), thus leading to a contradiction with our initial hypothesis. But first, let us introduce the following notations.

- Let \(J_{\text{max}} \in J\) be the (or any) job of \(J\) with the largest processing time.
- Let \(\mathcal{P}^{\text{max}}\) be the same job priority assignment as \(\mathcal{P}^{\text{other}}\), except that \(\mathcal{P}^{\text{max}}\) swaps the priority of \(J_{\text{low}}\) and \(J_{\text{max}}\), i.e., it assigns the lowest priority to the job \(J_{\text{max}}\) of largest processing time.
• Recall that $\overline{\text{comp}}^{(3)}(J, \pi, \mathcal{P}_{\text{other}})$ is an upper-bound on the completion time of the lowest priority job $j_{\text{low}}$ following $\mathcal{P}_{\text{other}}$. According to Expression 4.25 (page 356), $\overline{\text{comp}}^{(3)}(J, \pi, \mathcal{P}_{\text{other}})$ can be rewritten as

$$\overline{\text{comp}}^{(3)}_{\text{low}}(J, \pi, \mathcal{P}_{\text{other}}) \overset{\text{def}}{=} \mathcal{A}_{\text{low}}(J, \pi, \mathcal{P}_{\text{other}}) + \mathcal{B}_{\text{low}}(J, \pi, \mathcal{P}_{\text{other}}) \tag{4.34}$$

where

$$\mathcal{A}_{\text{low}}(J, \pi, \mathcal{P}_{\text{other}}) \overset{\text{def}}{=} \left( 1 - \frac{s_x}{\sum_{j=j} s_j^x} \right) \cdot \text{idle}_{m}(j_{\text{low}}, \pi, \mathcal{P}_{\text{other}}) \tag{4.35}$$

and

$$\mathcal{B}_{\text{low}}(J, \pi, \mathcal{P}_{\text{other}}) \overset{\text{def}}{=} c_{\text{low}} \frac{s_m}{s_x} + s_x \cdot \frac{\sum_{j \geq \text{other} \text{ low}} c_j}{s(1) \cdot \sum_{j=1} s_j} \tag{4.36}$$

• Similarly, $\overline{\text{comp}}^{(3)}(J, \pi, \mathcal{P}_{\text{max}})$ is an upper-bound on the completion time of the lowest priority job $j_{\text{max}}$ following $\mathcal{P}_{\text{max}}$ and $\overline{\text{comp}}^{(3)}(J, \pi, \mathcal{P}_{\text{max}})$ can be rewritten as

$$\overline{\text{comp}}^{(3)}_{\text{max}}(J, \pi, \mathcal{P}_{\text{max}}) \overset{\text{def}}{=} \mathcal{A}_{\text{max}}(J, \pi, \mathcal{P}_{\text{max}}) + \mathcal{B}_{\text{max}}(J, \pi, \mathcal{P}_{\text{max}}) \tag{4.37}$$

and

$$\mathcal{A}_{\text{max}}(J, \pi, \mathcal{P}_{\text{max}}) \overset{\text{def}}{=} \left( 1 - \frac{s_x}{\sum_{j=1} s_j^x} \right) \cdot \text{idle}_{m}(j_{\text{max}}, \pi, \mathcal{P}_{\text{max}}) \tag{4.38}$$

and

$$\mathcal{B}_{\text{max}}(J, \pi, \mathcal{P}_{\text{max}}) \overset{\text{def}}{=} c_{\text{max}} \frac{s_m}{s_x} + s_x \cdot \frac{\sum_{j \geq \text{max} \text{ max}} c_j}{s(1) \cdot \sum_{j=1} s_j} \tag{4.39}$$

In the following, we measure the difference between $\overline{\text{comp}}^{(3)}(J, \pi, \mathcal{P}_{\text{other}})$ and $\overline{\text{comp}}^{(3)}(J, \pi, \mathcal{P}_{\text{max}})$ and we show that

$$\overline{\text{comp}}^{(3)}_{\text{max}}(J, \pi, \mathcal{P}_{\text{max}}) \geq \overline{\text{comp}}^{(3)}_{\text{low}}(J, \pi, \mathcal{P}_{\text{other}})$$

thus leading to a contradiction with our initial assumption following which such a job priority assignment $\mathcal{P}_{\text{max}}$ does not exist. The remainder of the proof is divided into three parts. The first part measures the difference between $\mathcal{B}_{\text{low}}(J, \pi, \mathcal{P}_{\text{other}})$ and $\mathcal{B}_{\text{max}}(J, \pi, \mathcal{P}_{\text{max}})$, the second part measures the difference between $\mathcal{A}_{\text{low}}(J, \pi, \mathcal{P}_{\text{other}})$ and $\mathcal{A}_{\text{max}}(J, \pi, \mathcal{P}_{\text{max}})$ and finally, the third part asserts that $\overline{\text{comp}}^{(3)}_{\text{max}}(J, \pi, \mathcal{P}_{\text{max}}) \geq \overline{\text{comp}}^{(3)}_{\text{low}}(J, \pi, \mathcal{P}_{\text{other}})$. 

360
Part 1. According to Expressions 4.36 and 4.39, the difference between $B_{\max}(J, \pi, P_{\max})$ and $B_{\text{low}}(J, \pi, P_{\text{other}})$ is given by:

$$
B_{\max}(J, \pi, P_{\max}) - B_{\text{low}}(J, \pi, P_{\text{other}}) = \frac{c_{\text{max}} - c_{\text{low}}}{s_m} + s_x \cdot \frac{\sum_{j \geq \text{max}} s_j}{s(1) \cdot \sum_{j=1}^{x} s_j} - \frac{c_{\text{low}}}{s_m} - s_x \cdot \frac{\sum_{j \geq \text{other \, low}} c_j}{s(1) \cdot \sum_{j=1}^{x} s_j}
$$

Since the jobs $J_{\text{max}}$ and $J_{\text{low}}$ have the lowest priority according to $P_{\max}$ and $P_{\text{other}}$ (respectively) the above equality can be rewritten as

$$
B_{\max}(J, \pi, P_{\max}) - B_{\text{low}}(J, \pi, P_{\text{other}}) = \frac{c_{\text{max}} - c_{\text{low}}}{s_m} + \frac{s_x \cdot (c_{\text{low}} - c_{\text{max}})}{s(1) \cdot \sum_{j=1}^{x} s_j}
$$

$$
= \frac{s(1) \cdot \sum_{j=1}^{x} s_j \cdot (c_{\text{max}} - c_{\text{low}}) + s_m \cdot s_x \cdot (c_{\text{low}} - c_{\text{max}})}{s_m \cdot s(1) \cdot \sum_{j=1}^{x} s_j}
$$

$$
= \frac{s(1) \cdot \sum_{j=1}^{x} s_j \cdot c_{\text{max}} - c_{\text{low}} - s_m \cdot s_x \cdot c_{\text{max}} + c_{\text{low}}}{s_m \cdot s(1) \cdot \sum_{j=1}^{x} s_j}
$$

$$
= \frac{(s(1) \cdot \sum_{j=1}^{x} s_j - s_m \cdot s_x) \cdot (c_{\text{max}} - c_{\text{low}})}{s_m \cdot s(1) \cdot \sum_{j=1}^{x} s_j}
$$

Part 2. This part measures the difference between the terms $A_{\text{low}}(J, \pi, P_{\text{other}})$ and $A_{\text{max}}(J, \pi, P_{\text{max}})$. But first, recall that in the function $A_{\text{low}}(J, \pi, P_{\text{other}})$ defined in Expression 4.35, the term $\text{idle}_{m}(J_{\text{low}}, \pi, P_{\text{other}})$ denotes the $m^{th}$ idle-instant in the schedule of every job with a higher priority than $J_{\text{low}}$ following $P_{\text{other}}$, i.e., the schedule of $J \setminus \{J_{\text{low}}\}$. Similarly, in the function $A_{\text{max}}(J, \pi, P_{\text{max}})$ defined in Expression 4.38, the term $\text{idle}_{m}(J_{\text{max}}, \pi, P_{\text{max}})$ denotes the $m^{th}$ idle-instant in the schedule of every job with a higher priority than $J_{\text{max}}$ following $P_{\text{max}}$, i.e., the schedule of $J \setminus \{J_{\text{max}}\}$. Since $c_{\text{max}} > c_{\text{low}}$, we can consider that the set of jobs $J \setminus \{J_{\text{low}}\}$ is the same set of jobs as $J \setminus \{J_{\text{max}}\}$ except that the processing time $c_{\text{low}}$ of job $J_{\text{low}}$ (present in $J \setminus \{J_{\text{max}}\}$) has been increased to $c_{\text{max}}$ in $J \setminus \{J_{\text{low}}\}$. Therefore, it holds from Lemma 2.6 (page 109) that

$$
\text{idle}_{m}(J_{\text{low}}, \pi, P_{\text{other}}) \leq \text{idle}_{m}(J_{\text{max}}, \pi, P_{\text{max}}) + \frac{c_{\text{max}} - c_{\text{low}}}{s_m} \tag{4.40}
$$
As a result, according to Expression 4.35 and 4.38, the difference between $A_{\text{max}}(J, \pi, P^{\text{max}})$ and $A_{\text{low}}(J, \pi, P^{\text{other}})$ is given by:

\[
A_{\text{max}}(J, \pi, P^{\text{max}}) - A_{\text{low}}(J, \pi, P^{\text{other}}) = \left(1 - \frac{s_x}{\sum_{j=1}^{x} s_j}\right) \cdot \text{idle}_m(J_{\text{max}}, \pi, P^{\text{max}}) - \left(1 - \frac{s_x}{\sum_{j=1}^{x} s_j}\right) \cdot \text{idle}_m(J_{\text{low}}, \pi, P^{\text{other}})
\]

and from Inequality 4.40, the above equality can be rewritten as

\[
A_{\text{max}}(J, \pi, P^{\text{max}}) - A_{\text{low}}(J, \pi, P^{\text{other}}) \geq \left(1 - \frac{s_x}{\sum_{j=1}^{x} s_j}\right) \cdot \left(\text{idle}_m(J_{\text{max}}, \pi, P^{\text{max}}) - \text{idle}_m(J_{\text{low}}, \pi, P^{\text{other}})\right) - \frac{c_{\text{max}} - c_{\text{low}}}{s_m}
\]

Multiplying both sides by $(-1)$ yields

\[
A_{\text{low}}(J, \pi, P^{\text{other}}) - A_{\text{max}}(J, \pi, P^{\text{max}}) \leq \left(1 - \frac{s_x}{\sum_{j=1}^{x} s_j}\right) \cdot \frac{c_{\text{max}} - c_{\text{low}}}{s_m} \leq \frac{\sum_{j=1}^{x} s_j - s_x}{s_m \cdot \sum_{j=1}^{x} s_j} \cdot (c_{\text{max}} - c_{\text{low}})
\]

**Part 3.** This third part asserts that $\overline{\text{comp}}^{\text{(3)}}_{\text{max}}(J, \pi, P^{\text{max}}) \geq \overline{\text{comp}}^{\text{(3)}}_{\text{low}}(J, \pi, P^{\text{other}})$. Recall that from Parts 1 and 2 we have:

\[
B_{\text{max}}(J, \pi, P^{\text{max}}) = B_{\text{low}}(J, \pi, P^{\text{other}}) + \frac{(s(1) \cdot \sum_{j=1}^{x} s_j - s_m \cdot s_x) \cdot (c_{\text{max}} - c_{\text{low}})}{s_m \cdot s(1) \cdot \sum_{j=1}^{x} s_j}
\]

\[
A_{\text{max}}(J, \pi, P^{\text{max}}) \geq A_{\text{low}}(J, \pi, P^{\text{other}}) - \frac{(\sum_{j=1}^{x} s_j - s_x) \cdot (c_{\text{max}} - c_{\text{low}})}{s_m \cdot \sum_{j=1}^{x} s_j}
\]

From the definitions of $\overline{\text{comp}}^{\text{(3)}}_{\text{max}}(J, \pi, P^{\text{max}})$ and $\overline{\text{comp}}^{\text{(3)}}_{\text{low}}(J, \pi, P^{\text{other}})$ in Expression 4.37 and 4.34, we have

\[
\overline{\text{comp}}^{\text{(3)}}_{\text{max}}(J, \pi, P^{\text{max}}) - \overline{\text{comp}}^{\text{(3)}}_{\text{low}}(J, \pi, P^{\text{other}}) = A_{\text{max}}(J, \pi, P^{\text{max}}) + B_{\text{max}}(J, \pi, P^{\text{max}}) - A_{\text{low}}(J, \pi, P^{\text{other}}) - B_{\text{low}}(J, \pi, P^{\text{other}})
\]

By replacing $B_{\text{max}}(J, \pi, P^{\text{max}})$ with $B_{\text{low}}(J, \pi, P^{\text{other}}) + \frac{(s(1) \cdot \sum_{j=1}^{x} s_j - s_m \cdot s_x) \cdot (c_{\text{max}} - c_{\text{low}})}{s_m \cdot s(1) \cdot \sum_{j=1}^{x} s_j}$ according to Equality 4.41, we get
\[
\overline{\text{comp}}^{(3)}_{\text{max}}(J, \pi, \mathcal{P}^{\text{max}}) - \overline{\text{comp}}^{(3)}_{\text{low}}(J, \pi, \mathcal{P}^{\text{other}})
\]

\[
= \mathcal{A}_{\text{max}}(J, \pi, \mathcal{P}^{\text{max}}) - \mathcal{A}_{\text{low}}(J, \pi, \mathcal{P}^{\text{other}}) + \frac{(s(1) \cdot \sum_{j=1}^{x} s_j - s_m \cdot s_x) \cdot (c_{\text{max}} - c_{\text{low}})}{s_m \cdot s(1) \cdot \sum_{j=1}^{x} s_j}
\]

and replacing \(\mathcal{A}_{\text{max}}(J, \pi, \mathcal{P}^{\text{max}})\) with \(\mathcal{A}_{\text{low}}(J, \pi, \mathcal{P}^{\text{other}}) - \frac{(\sum_{j=1}^{x} s_j - s_m \cdot s_x) \cdot (c_{\text{max}} - c_{\text{low}})}{s_m \cdot \sum_{j=1}^{x} s_j}\) according to Inequality 4.42 yields \(\overline{\text{comp}}^{(3)}_{\text{max}}(J, \pi, \mathcal{P}^{\text{max}}) - \overline{\text{comp}}^{(3)}_{\text{low}}(J, \pi, \mathcal{P}^{\text{other}})\)

\[
\overline{\text{comp}}^{(3)}_{\text{max}}(J, \pi, \mathcal{P}^{\text{max}}) - \overline{\text{comp}}^{(3)}_{\text{low}}(J, \pi, \mathcal{P}^{\text{other}}) - \frac{(s(1) \cdot \sum_{j=1}^{x} s_j - s_m \cdot s_x) \cdot (c_{\text{max}} - c_{\text{low}})}{s_m \cdot s(1) \cdot \sum_{j=1}^{x} s_j}
\]

\[
= \frac{-s_m s_x c_{\text{max}} + s_m s_x c_{\text{low}} + s(1) s_x c_{\text{max}} - s(1) s_x c_{\text{low}}}{s_m \cdot s(1) \cdot \sum_{j=1}^{x} s_j}
\]

\[
= \frac{s_m s_x \cdot (c_{\text{low}} - c_{\text{max}}) + s(1) s_x \cdot (c_{\text{max}} - c_{\text{low}})}{s_m \cdot s(1) \cdot \sum_{j=1}^{x} s_j}
\]

\[
= \frac{s(1) s_x \cdot (c_{\text{max}} - c_{\text{low}}) - s_m s_x \cdot (c_{\text{max}} - c_{\text{low}})}{s_m \cdot s(1) \cdot \sum_{j=1}^{x} s_j}
\]

\[
= \frac{s_x \cdot (s(1) - s_m) \cdot (c_{\text{max}} - c_{\text{low}})}{s_m \cdot s(1) \cdot \sum_{j=1}^{x} s_j}
\]

\[
= 0
\]

In conclusion, \(\mathcal{P}^{\text{max}}\) is a job priority assignment that assigns the lowest priority to the job \(J_{\text{max}}\) with the largest processing time and we showed in this third part that

\[
\overline{\text{comp}}^{(3)}_{\text{max}}(J, \pi, \mathcal{P}^{\text{max}}) \geq \overline{\text{comp}}^{(3)}_{\text{low}}(J, \pi, \mathcal{P}^{\text{other}})
\]

Since by hypothesis we know that \(\overline{\text{comp}}^{(3)}_{\text{low}}(J, \pi, \mathcal{P}^{\text{other}})\) is an upper-bound on the maximum makespan, it holds that \(\overline{\text{comp}}^{(3)}_{\text{max}}(J, \pi, \mathcal{P}^{\text{max}})\) is also an upper-bound on the maximum makespan. This contradicts our initial assumption following which such a job priority assignment \(\mathcal{P}^{\text{max}}\) does not exists.
Appendix

Lemma 4.16

Suppose that $c_1 \leq c_2 \leq \cdots \leq c_n$. If $J$ is scheduled upon $\pi$ by any global, FJP and strongly work-conserving scheduler, then an upper-bound $\hat{m}_{\text{unif}}(J, \pi)$ on the makespan is given by $\hat{t}_n$, where $\hat{t}_n$ is computed by the following iterative process.

\[
\begin{align*}
\hat{t}_1 &= \frac{c_1}{s_m} \quad \text{(initialization)} \\
\hat{t}_{i+1} &= \left(1 - \frac{s_x}{\sum_{j=i}^x s_j}\right) \cdot \hat{t}_i + \left(\frac{c_{i+1}}{s_m} + s_x \cdot \frac{\sum_{j=1}^{i} c_j}{s(1) \cdot \sum_{j=1}^{x} s_j}\right) \quad \text{(iterative step)}
\end{align*}
\]

Proof

From Lemma 4.15, we know that there exists at least one job priority assignment $\mathcal{P}_{\text{max}}$ that assigns the lowest priority to the job $J_n$ with the largest processing time and such that

\[\text{comp}_n^{(3)}(J, \pi, \mathcal{P}_{\text{max}}) \geq \text{maximum makespan}\]

From Expression 4.25 (on page 356) we know that

\[\text{comp}_n^{(3)}(J, \pi, \mathcal{P}_{\text{max}}) \overset{\text{def}}{=} \left(1 - \frac{s_x}{\sum_{j=i}^x s_j}\right) \cdot \text{idle}_m(J_n, \mathcal{P}) + \left(\frac{c_n}{s_m} + s_x \cdot \frac{\sum_{j=1}^{n} c_j}{s(1) \cdot \sum_{j=1}^{x} s_j}\right)\]

and it obviously holds $\forall i \in [1, n]$ that the exact makespan generated by the set of jobs $J_j >_{\mathcal{P}_{\text{max}}} J_i$ cannot be larger than the maximum completion time of these jobs $J_j >_{\mathcal{P}_{\text{max}}} J_i$. That is, it holds $\forall i \in [1, n]$ that

\[\text{idle}_m(J_i, \pi, \mathcal{P}_{\text{max}}) \leq \max_{J_j >_{\mathcal{P}_{\text{max}}} J_i} \left\{\text{comp}_n^{(3)}(J, \pi, \mathcal{P}_{\text{max}})\right\}\]

Consequently, Expression 4.43 can be rewritten as

\[\text{comp}_n^{(3)}(J, \pi, \mathcal{P}_{\text{max}}) \leq \left(1 - \frac{s_x}{\sum_{j=i}^x s_j}\right) \cdot \max_{J_j >_{\mathcal{P}_{\text{max}}} J_i} \left\{\text{comp}_n^{(3)}(J, \pi, \mathcal{P})\right\} + \left(\frac{c_n}{s_m} + s_x \cdot \frac{\sum_{j=1}^{n} c_j}{s(1) \cdot \sum_{j=1}^{x} s_j}\right)\]

and an upper-bound on $\text{comp}_n^{(3)}(J, \pi, \mathcal{P}_{\text{max}})$ is thus given by $\hat{t}_n$, where

\[\hat{t}_n \overset{\text{def}}{=} \left(1 - \frac{s_x}{\sum_{j=i}^x s_j}\right) \cdot \max_{J_j >_{\mathcal{P}_{\text{max}}} J_i} \left\{\text{comp}_n^{(3)}(J, \pi, \mathcal{P})\right\} + \left(\frac{c_n}{s_m} + s_x \cdot \frac{\sum_{j=1}^{n} c_j}{s(1) \cdot \sum_{j=1}^{x} s_j}\right)\]

Considering the subset $J' = \{J_1, J_2, \ldots, J_{n-1}\}$ of jobs, we know from Lemma 4.15 that there exists at least one job priority assignment $\mathcal{P}'$ that assigns the lowest priority to the job $J_{n-1}$ with the largest processing time and such that $\forall X$
\[
\overline{\text{comp}}_{n-1}^{(3)}(J', \pi, P') = \max_{i=1}^{n-1} \{ \overline{\text{comp}}_i^{(3)}(J', \pi, X) \}
\]
i.e., \(\overline{\text{comp}}_{n-1}^{(3)}(J', \pi, P')\) is an upper-bound on the makespan for the set \(J'\) of jobs, while considering every job priority assignments \(X\). Thereby, if \(P^{\text{max}}\) assigns a priority to \(J_{n-1}\) such that \(\forall i \in [1, n-2]\),
\[
J_i > P^{\text{max}} J_{n-1} > P^{\text{max}} J_n
\]
then we get
\[
\overline{\text{comp}}_{n-1}^{(3)}(J', \pi, P^{\text{max}}) = \max_{i=1}^{n-1} \{ \overline{\text{comp}}_i^{(3)}(J', \pi, P^{\text{max}}) \}
\]
and Expression 4.45 can be rewritten as
\[
\hat{t}_n \overset{\text{def}}{=} \left(1 - \frac{s_x}{\sum_{j=1}^{s} s_j} \right) \cdot \overline{\text{comp}}_{n-1}^{(3)}(J, \pi, P) + \left( \frac{c_n}{s_m} + s_x \cdot \frac{\sum_{j=1}^{s_{j>\pi}} c_j}{s(1) \cdot \sum_{j=1}^{s} s_j} \right)
\]
The same reasoning as the one above can be used iteratively in order to determine an upper-bound on \(\overline{\text{comp}}_{n-1}^{(3)}(J', \pi, P^{\text{max}})\). Ultimately, this iterative development causes \(P^{\text{max}}\) to imitate the job priority assignment SJF (Shortest Job First) and an upper-bound \(\overline{m}^{\text{unif}}_{s_3}(J, \pi)\) on the makespan can therefore be computed by the iterative process:
\[
\begin{cases}
\hat{t}_1 = \frac{c_1}{s_m} & \text{(initialization)} \\
\hat{t}_{i+1} = \left(1 - \frac{s_x}{\sum_{j=1}^{s} s_j} \right) \cdot \hat{t}_i + \left( \frac{c_{i+1}}{s_m} + s_x \cdot \frac{\sum_{j=1}^{i} c_j}{s(1) \cdot \sum_{j=1}^{s} s_j} \right) & \text{(iterative step)}
\end{cases}
\]
That is, even though we know that scheduling the jobs according to SJF does not lead to the maximum makespan (as showed in the counterexample of Section 2.8.1, page 129), this policy leads nevertheless to an upper-bound \(\overline{m}^{\text{unif}}_{s_3}(J, \pi)\) on the upper-bounds \(\overline{\text{comp}}_{n-1}^{(3)}(J, \pi, X) \forall X\). The lemma follows.

**Lemma 4.17**

Suppose that \(c_1 \leq c_2 \leq \cdots \leq c_n\). If \(J\) is scheduled upon \(\pi\) by any FJP job priority assignment then an upper-bound \(\overline{m}^{\text{unif}}_{s_3}(J, \pi)\) on the makespan is given by

365
Appendix

\[ \text{ms} = \frac{1}{n} \sum_{\ell=1}^{n} \left( \frac{c_{\ell} + s_{x} \cdot s_{m}}{s(1) \cdot \sum_{j=1}^{n} s_{j}} \cdot \sum_{\ell-1}^{n} e_{j} \right) \cdot H_{n-\ell} \]  \hspace{1cm} (4.46)

where \( H_{j} \) is such that \( \forall j, \)

\[ H_{j} \overset{\text{def}}{=} \begin{cases} 1 & \text{if } s_{x} = \sum_{i=1}^{x} s_{i} \text{ and } j = 0 \\ \left(1 - \frac{s_{x}}{\sum_{i=1}^{x} s_{i}}\right)^{j} & \text{otherwise} \end{cases} \]  \hspace{1cm} (4.49)

Proof

This proof consists in rewriting the recursive process given in Lemma 4.16 as a non-recursive function. First, we rewrite the iterative process as follows.

\[ \begin{align*}
\hat{t}_{1} &= \frac{c_{1}}{s_{m}} \quad \text{(initialization)} \\
\hat{t}_{i} &= H \cdot \hat{t}_{i-1} + Q_{i} \quad \text{(iterative step)}
\end{align*} \]

where \( H \overset{\text{def}}{=} \left(1 - \frac{s_{x}}{\sum_{j=1}^{n} s_{j}}\right) \) and \( Q_{i} \overset{\text{def}}{=} \frac{c_{i}}{s_{m}} + s_{x} \cdot \frac{\sum_{i=1}^{n} e_{i}}{s(1) \sum_{j=1}^{n} s_{j}}. \) From the iterative step of this simplified process, we have \( \forall i, 2 \leq i \leq n, \)

\[ \hat{t}_{i} = H \cdot \hat{t}_{i-1} + Q_{i} = H \cdot (H \cdot \hat{t}_{i-2} + Q_{i-1}) + Q_{i} = H \cdot (H \cdot (H \cdot \hat{t}_{i-3} + Q_{i-2}) + Q_{i-1}) + Q_{i} = H \cdot (H \cdot \ldots \cdot H \cdot (\hat{t}_{1} + Q_{2}) + \ldots + Q_{i-2}) + Q_{i-1}) + Q_{i} \]

\[ = H^{i-1} \cdot \hat{t}_{1} + \sum_{j=0}^{i-2} H^{j} \cdot Q_{i-j} \]  \hspace{1cm} (4.48)

Notice that in the sum above, we get the term \( H^{0} + Q_{i} \) when \( j = 0 \) and according to the definition of \( H \overset{\text{def}}{=} \left(1 - \frac{s_{x}}{\sum_{j=1}^{n} s_{j}}\right), \) in the uniprocessor case where \( s_{x} = \sum_{j=1}^{n} s_{j}, \) we get \( H^{0} = (1 - 1)^{0} \) which is undetermined. Thereby, passing from Expression 4.47 to Expression 4.48 requires to redefine \( H \) beforehand, i.e., we define \( H_{j} \forall j \) such that

\[ H_{j} \overset{\text{def}}{=} \begin{cases} 1 & \text{if } s_{x} = \sum_{i=1}^{n} s_{i} \text{ and } j = 0 \\ \left(1 - \frac{s_{x}}{\sum_{i=1}^{n} s_{i}}\right)^{j} & \text{otherwise} \end{cases} \]  \hspace{1cm} (4.49)
and Equality 4.48 can be rewritten as

\[ \hat{t}_i = H_{i-1} \cdot \hat{t}_1 + \sum_{j=0}^{i-2} H_j \cdot Q_{i-j} \]

Let \( \ell \overset{\text{def}}{=} i - j \), the previous equality yields

\[
\hat{t}_i = H_{i-1} \cdot \hat{t}_1 + \sum_{\ell=2}^{i} H_{i-\ell} \cdot Q_{\ell}
\]

\[
= H_{i-1} \cdot \frac{c_1}{s_m} + \sum_{\ell=2}^{i} H_{i-\ell} \cdot \left( \frac{c_\ell}{s_m} + s_x \cdot \frac{\sum_{j=1}^{\ell-1} c_j}{s(1) \cdot \sum_{j=1}^{s} j} \right)
\]

\[
= \frac{1}{s_m} \cdot \left( c_1 \cdot H_{i-1} + \sum_{\ell=2}^{i} \left( c_\ell + \frac{s_x \cdot s_m}{s(1) \cdot \sum_{j=1}^{s} j} \cdot \sum_{j=1}^{\ell-1} c_j \right) \cdot H_{i-\ell} \right)
\]

\[
= \frac{1}{s_m} \cdot \sum_{\ell=1}^{i} \left( c_\ell + \frac{s_x \cdot s_m}{s(1) \cdot \sum_{j=1}^{s} j} \cdot \sum_{j=1}^{\ell-1} c_j \right) \cdot H_{i-\ell} \quad \text{since} \quad \frac{s_x \cdot s_m}{s(1) \cdot \sum_{j=1}^{s} j} \cdot \sum_{j=1}^{\ell-1} c_j = 0 \quad \text{for} \quad \ell = 1
\]

That is, for \( i = n \) we have

\[
\hat{t}_n = \frac{1}{s_m} \cdot \sum_{\ell=1}^{n} \left( c_\ell + \frac{s_x \cdot s_m}{s(1) \cdot \sum_{j=1}^{s} j} \cdot \sum_{j=1}^{\ell-1} c_j \right) \cdot H_{n-\ell}
\]  

(4.50)

where \( H \) is such as defined in Expression 4.49. The lemma follows from the fact that \( \overline{m}^{\text{uni}}(J, \pi) \overset{\text{def}}{=} \hat{t}_n \) from Lemma 4.16.

From Lemma 4.17, a sufficient validity test can therefore be formalized as follows for the protocol SM-MSO.

**Validity Test 4.1 (For SM-MSO, FJP schedulers and uniform platforms)**

For any multi-mode real-time application \( \tau \) and any identical platform \( \pi \) composed of \( m \) processors, the protocol SM-MSO is valid provided that, for every mode \( M^i \),

\[
\overline{m}^{\text{uni}}(J^{wc}, \pi) \leq \min_{j \neq i} \left\{ \min_{k=1}^{n_i} \left\{ D_k(M^i) \right\} \right\}
\]
where $\mathcal{J}^\text{wc}_i$ is the worst-case rem-job set issued from mode $M_i$. This set contains $n_i$ jobs of processing time $C_1^i, C_2^i, \ldots, C_{n_i}^i$ and is ordered by non-decreasing order of job processing times, i.e., $C_j^i \geq C_{j-1}^i \forall j = 2, 3, \ldots, n_i$. 
References


(a) Visualization of the equilibrium instant equi. Informally, this instant equi is such that the amounts of execution units executed in the yellow and red areas are equivalent.

(b) Example of a possible shape of the yellow area. This shape can be divided into at most \((m - 1)\) rectangles \(R_1, R_2, \ldots, R_{m-1}\) such that the height of every rectangle \(R_j\) (with \(j = 2, \ldots, m - 1\)) is higher than that of the rectangle \(R_{j-1}\).

**Figure 4.21:** Representation of the different notations used in the proof.
Appendix

Average relative consumption of I−EDF−MAX, I−EDF and I−EDF(k) for Dmax in [0.1, 1]

(a) The X-axis ranges the values of $D_{\text{max}}$ from 0.1 to 1. The Y-axis displays the relative energy consumption (in %); relative to the consumption of CLV, i.e., the consumption of CLV is 100 %. The height of the rectangles gives the average relative consumption of each method (I-EDF$^{\text{max}}$ in red, I-EDF in blue and I-EDF$^{(k)}$ in green).

(b) The X-axis ranges the values of $D_{\text{max}}$ from 0.1 to 1. The Y-axis displays the relative energy consumption (in %); relative to the consumption of CLV, i.e., the consumption of CLV is 100 %. Each box plot provides (from up to bottom) the maximal recorded consumption, the upper quartile, the median, the lower quartile and the minimal recorded consumption.

Figure 4.22: Some statistics about the consumption of I-EDF$^{\text{max}}$ (in red), I-EDF (in blue) and I-EDF$^{(k)}$ (in green), simulated on Transmeta Crusoe TM5400 processors. In both figures, the X-axis displays every value of $D_{\text{max}}$ while the Y-axis gives the relative consumption of each method (relative to the consumption of CLV).
Appendix

(a) The X-axis ranges the values of $D_{\text{max}}$ from 0.1 to 1. The Y-axis displays the relative energy consumption (in %); relative to the consumption of CLV, i.e., the consumption of CLV is 100 %. The height of each rectangle gives the average relative consumption of each method.

(b) The X-axis ranges the values of $D_{\text{max}}$ from 0.1 to 1. The Y-axis displays the relative energy consumption (in %); relative to the consumption of CLV, i.e., the consumption of CLV is 100 %. Each box plot provides (from up to bottom) the maximal recorded consumption, the upper quartile, the median, the lower quartile and the minimal recorded consumption.

Figure 4.23: Some statistics about the consumption of $I$-$EDF^{\text{max}}$ (in red), $I$-$EDF^{(k)}$ (in glowing green), $\text{MORA-EDF}^{(k)}$ (in white), $\text{MOTE-EDF}^{(k)}$ (in gray) and $\text{MORAOTE-EDF}^{(k)}$ (in gold), simulated on Transmeta Crusoe TM5400 processors. In both figures, the X-axis displays every value of $D_{\text{max}}$ while the Y-axis gives the relative consumption of each method (relative to the consumption of CLV).
Appendix

(a) The X-axis ranges the values of $D_{\text{max}}$ from 0.1 to 1. The Y-axis displays the relative energy consumption (in %); relative to the consumption of CLV, i.e., the consumption of CLV is 100 %. The height of the rectangles gives the average relative consumption of each method (I-EDF$^{\text{max}}$ in red, I-EDF in blue and I-EDF$^{(k)}$ in green).

(b) The X-axis ranges the values of $D_{\text{max}}$ from 0.1 to 1. The Y-axis displays the relative energy consumption (in %); relative to the consumption of CLV, i.e., the consumption of CLV is 100 %. Each box plot provides (from up to bottom) the maximal recorded consumption, the upper quartile, the median, the lower quartile and the minimal recorded consumption.

**Figure 4.24:** Some statistics about the consumption of I-EDF$^{\text{max}}$ (in red), I-EDF (in blue) and I-EDF$^{(k)}$ (in green), simulated on PowerPC-405LP processors. In both figures, the X-axis displays every value of $D_{\text{max}}$ while the Y-axis gives the relative consumption of each method (relative to the consumption of CLV).
Appendix

Average relative consumption of I-EDF-MAX, I-EDF(k), MORA-EDF(k), MOTE-EDF(k) and MORAOTE-EDF(k) for Dmax in [0.1, 1]

(a) The X-axis ranges the values of Dmax from 0.1 to 1. The Y-axis displays the relative energy consumption (in %); relative to the consumption of CLV, i.e., the consumption of CLV is 100%. The height of each rectangle gives the average relative consumption of each method.

(b) The X-axis ranges the values of Dmax from 0.1 to 1. The Y-axis displays the relative energy consumption (in %); relative to the consumption of CLV, i.e., the consumption of CLV is 100%. Each box plot provides (from up to bottom) the maximal recorded consumption, the upper quartile, the median, the lower quartile and the minimal recorded consumption.

Figure 4.25: Some statistics about the consumption of I-EDF<sup>max</sup> (in red), I-EDF<sup>(k)</sup> (in glowing green), MORA-EDF<sup>(k)</sup> (in white), MOTE-EDF<sup>(k)</sup> (in gray) and MORAOTE-EDF<sup>(k)</sup> (in gold), simulated on PowerPC-405LP processors. In both figures, the X-axis displays every value of Dmax while the Y-axis gives the relative consumption of each method (relative to the consumption of CLV).
Appendix

**Average relative consumption of I−EDF−MAX, I−EDF and I−EDF(k) for Dmax in [0.1, 1]**

(a) The X-axis ranges the values of $D_{\text{max}}$ from 0.1 to 1. The Y-axis displays the relative energy consumption (in %); relative to the consumption of CLV, i.e., the consumption of CLV is 100 %. The height of the rectangles gives the average relative consumption of each method (I-EDF$^{\text{max}}$ in red, I-EDF in blue and I-EDF$^{(k)}$ in green).

**Relative consumption of I−EDF−MAX, I−EDF and I−EDF(k) for Dmax in [0.1, 1]**

(b) The X-axis ranges the values of $D_{\text{max}}$ from 0.1 to 1. The Y-axis displays the relative energy consumption (in %); relative to the consumption of CLV, i.e., the consumption of CLV is 100 %. Each box plot provides (from up to bottom) the maximal recorded consumption, the upper quartile, the median, the lower quartile and the minimal recorded consumption.

**Figure 4.26:** Some statistics about the consumption of I-EDF$^{\text{max}}$ (in red), I-EDF (in blue) and I-EDF$^{(k)}$ (in green), simulated on Intel PXA270 processors. In both figures, the X-axis displays every value of $D_{\text{max}}$ while the Y-axis gives the relative consumption of each method (relative to the consumption of CLV).
Appendix

(a) The X-axis ranges the values of $D_{\text{max}}$ from 0.1 to 1. The Y-axis displays the relative energy consumption (in %); relative to the consumption of CLV, i.e., the consumption of CLV is 100%. The height of each rectangle gives the average relative consumption of each method, relatively to the consumption of CLV.

(b) The X-axis ranges the values of $D_{\text{max}}$ from 0.1 to 1. The Y-axis displays the relative energy consumption (in %); relative to the consumption of CLV, i.e., the consumption of CLV is 100%. Each box plot provides (from up to bottom) the maximal recorded consumption, the upper quartile, the median, the lower quartile and the minimal recorded consumption.

Figure 4.27: Some statistics about the consumption of $I\text{-EDF}^\text{max}$ (in red), $I\text{-EDF}(k)$ (in glowing green), MORA-EDF$(k)$ (in white), MOTE-EDF$(k)$ (in gray) and MORAOTE-EDF$(k)$ (in gold), simulated on Intel PXA270 processors. In both figures, the X-axis displays every value of $D_{\text{max}}$ while the Y-axis gives the relative consumption of each method (relative to the consumption of CLV).
(a) The X-axis ranges the values of $D_{\text{max}}$ from 0.1 to 1. The Y-axis displays the relative energy consumption (in %); relative to the consumption of CLV, i.e., the consumption of CLV is 100 %. The height of the rectangles gives the average relative consumption of each method ($I$-EDF$^{\text{max}}$ in red, $I$-EDF in blue and $I$-EDF$^{(k)}$ in green).

(b) The X-axis ranges the values of $D_{\text{max}}$ from 0.1 to 1. The Y-axis displays the relative energy consumption (in %); relative to the consumption of CLV, i.e., the consumption of CLV is 100 %. Each box plot provides (from up to bottom) the maximal recorded consumption, the upper quartile, the median, the lower quartile and the minimal recorded consumption.

Figure 4.28: Some statistics about the consumption of $I$-EDF$^{\text{max}}$ (in red), $I$-EDF (in blue) and $I$-EDF$^{(k)}$ (in green), simulated on Intel XScale processors. In both figures, the X-axis displays every value of $D_{\text{max}}$ while the Y-axis gives the relative consumption of each method (relative to the consumption of CLV).
Appendix

Average relative consumption of I-EDF-MAX, I-EDF(k), MORA-EDF(k), MOTE-EDF(k) and MORAOTE-EDF(k) for Dmax in [0.1, 1]

(a) The X-axis ranges the values of D_max from 0.1 to 1. The Y-axis displays the relative energy consumption (in %); relative to the consumption of CLV, i.e., the consumption of CLV is 100 %. The height of each rectangle gives the average relative consumption of each method, relatively to the consumption of CLV.

Relative consumption of I-EDF-MAX, I-EDF(k), MORA-EDF(k), MOTE-EDF(k) and MORAOTE-EDF(k) for Dmax in [0.1, 1]

(b) The X-axis ranges the values of D_max from 0.1 to 1. The Y-axis displays the relative energy consumption (in %); relative to the consumption of CLV, i.e., the consumption of CLV is 100 %. Each box plot provides (from up to bottom) the maximal recorded consumption, the upper quartile, the median, the lower quartile and the minimal recorded consumption.

Figure 4.29: Some statistics about the consumption of I-EDF\textsuperscript{max} (in red), I-EDF\textsuperscript{k} (in glowing green), MORA-EDF\textsuperscript{k} (in white), MOTE-EDF\textsuperscript{k} (in gray) and MORAOTE-EDF\textsuperscript{k} (in gold), simulated on Intel XScale processors. In both figures, the X-axis displays every value of D_max while the Y-axis gives the relative consumption of each method (relative to the consumption of CLV).
**Table of symbols**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau$</td>
<td>12</td>
<td>An application.</td>
</tr>
<tr>
<td>$\tau_i$</td>
<td>12</td>
<td>The $i$th task of $\tau$.</td>
</tr>
<tr>
<td>$D_i$</td>
<td>12</td>
<td>The relative deadline of the task $\tau_i$.</td>
</tr>
<tr>
<td>$T_i$</td>
<td>12</td>
<td>The period of the task $\tau_i$.</td>
</tr>
<tr>
<td>$C_i$</td>
<td>13</td>
<td>The worst-case execution time of the task $\tau_i$.</td>
</tr>
<tr>
<td>$O_i$</td>
<td>14</td>
<td>The offset of the task $\tau_i$.</td>
</tr>
<tr>
<td>$\tau_{i,j}$</td>
<td>15</td>
<td>The $j$th job of the task $\tau_i$.</td>
</tr>
<tr>
<td>$a_{i,j}$</td>
<td>15</td>
<td>The release time of the $j$th job of the task $\tau_i$.</td>
</tr>
<tr>
<td>$d_{i,j}$</td>
<td>15</td>
<td>The absolute deadline of the $j$th job of the task $\tau_i$.</td>
</tr>
<tr>
<td>run($\tau, t$)</td>
<td>15</td>
<td>The subset of running tasks in $\tau$.</td>
</tr>
<tr>
<td>wait($\tau, t$)</td>
<td>15</td>
<td>The subset of waiting tasks in $\tau$.</td>
</tr>
<tr>
<td>active($\tau, t$)</td>
<td>15</td>
<td>The subset of active tasks in $\tau$.</td>
</tr>
<tr>
<td>$U_i$</td>
<td>16</td>
<td>The utilization of the task $\tau_i$.</td>
</tr>
<tr>
<td>$U_{\text{sum}}(\tau)$</td>
<td>16</td>
<td>The generalized utilization of the set of tasks $\tau$.</td>
</tr>
<tr>
<td>$U_{\max}(\tau)$</td>
<td>16</td>
<td>The maximal utilization of the set of tasks $\tau$.</td>
</tr>
<tr>
<td>$\delta_i$</td>
<td>17</td>
<td>The density of the task $\tau_i$.</td>
</tr>
<tr>
<td>$\delta_{\text{sum}}(\tau)$</td>
<td>17</td>
<td>The generalized density of the set of tasks $\tau$.</td>
</tr>
<tr>
<td>$\delta_{\max}(\tau)$</td>
<td>17</td>
<td>The maximal density of the set of tasks $\tau$.</td>
</tr>
</tbody>
</table>
### Table of symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{DBF}(\tau_i, t)$</td>
<td>17</td>
<td>The demand bound function of the task $\tau_i$ in a time interval of length $t$.</td>
</tr>
<tr>
<td>$\text{LOAD}(\tau)$</td>
<td>17</td>
<td>The load of the task set $\tau$.</td>
</tr>
<tr>
<td>$M^k$</td>
<td>20</td>
<td>The $k^{th}$ mode of operation of the application $\tau$.</td>
</tr>
<tr>
<td>$\tau^k$</td>
<td>20</td>
<td>The set of tasks of mode $M^k$.</td>
</tr>
<tr>
<td>$n_k$</td>
<td>20</td>
<td>The number of tasks in $\tau^k$.</td>
</tr>
<tr>
<td>$\tau^k_i$</td>
<td>20</td>
<td>The $i^{th}$ task of $\tau^k$.</td>
</tr>
<tr>
<td>$D^k_i$</td>
<td>77</td>
<td>The <em>relative</em> deadline of the task $\tau^k_i$.</td>
</tr>
<tr>
<td>$T^k_i$</td>
<td>77</td>
<td>The period of the task $\tau^k_i$.</td>
</tr>
<tr>
<td>$C^k_i$</td>
<td>77</td>
<td>The worst-case execution time of the task $\tau^k_i$.</td>
</tr>
<tr>
<td>$\tau^k_{i,j}$</td>
<td>79</td>
<td>The $j^{th}$ job of the task $\tau^k_i$.</td>
</tr>
<tr>
<td>$d^k_{i,j}$</td>
<td>79</td>
<td>The absolute deadline of the job $\tau^k_{i,j}$.</td>
</tr>
<tr>
<td>$\text{enabled}(\tau^k, t)$</td>
<td>78</td>
<td>The subset of enabled tasks of $\tau^k$ at time $t$.</td>
</tr>
<tr>
<td>$\text{disabled}(\tau^k, t)$</td>
<td>78</td>
<td>The subset of disabled tasks of $\tau^k$ at time $t$.</td>
</tr>
<tr>
<td>$\pi$</td>
<td>78</td>
<td>A multiprocessor platform.</td>
</tr>
<tr>
<td>$m$</td>
<td>78</td>
<td>The number of processors in $\pi$.</td>
</tr>
<tr>
<td>$\pi_i$</td>
<td>78</td>
<td>The $i^{th}$ processor of the platform $\pi$.</td>
</tr>
<tr>
<td>$s_i$</td>
<td>78</td>
<td>The computing capacity (i.e., the speed) of the processor $\pi_i$.</td>
</tr>
<tr>
<td>$s(k)$</td>
<td>78</td>
<td>The cumulated speeds of the $(m - k + 1)$ fastest processors of $\pi$.</td>
</tr>
<tr>
<td>$\text{MCR}(j)$</td>
<td>79</td>
<td>A mode change request to mode $M^j$.</td>
</tr>
<tr>
<td>$t_{\text{MCR}(j)}$</td>
<td>79</td>
<td>The <em>latest</em> invoking time of a MCR$(j)$.</td>
</tr>
</tbody>
</table>
**Table of symbols**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_k(M')$</td>
<td>79</td>
<td>The relative mode change deadline of the task $\tau_k^l$ during every transition from the mode $M'$ to the mode $M'^l$.</td>
</tr>
<tr>
<td>$S^k$</td>
<td>81</td>
<td>The scheduling algorithm used by the mode $M^k$.</td>
</tr>
<tr>
<td>$J$</td>
<td>82</td>
<td>A finite set of jobs.</td>
</tr>
<tr>
<td>$J_i$</td>
<td>82</td>
<td>The $i^{th}$ job of $J$.</td>
</tr>
<tr>
<td>$c_i$</td>
<td></td>
<td>The execution time of job $J_i$.</td>
</tr>
<tr>
<td>$J_i &gt;_{S^k} J_j$</td>
<td>83</td>
<td>$J_i$ has a higher priority than $J_j$ according to the scheduler $S^k$.</td>
</tr>
<tr>
<td>$P$</td>
<td>83</td>
<td>A job priority assignment.</td>
</tr>
<tr>
<td>$J_i &gt;_P J_j$</td>
<td>83</td>
<td>$J_i$ has a higher priority than $J_j$ according to $P$.</td>
</tr>
<tr>
<td>$J_{wc}^i$</td>
<td>89</td>
<td>The worst-case rem-job set of mode $M^i$.</td>
</tr>
<tr>
<td>$S_{trans}$</td>
<td>90</td>
<td>The scheduler used by AM-MSO during any mode transition.</td>
</tr>
<tr>
<td>$M^{old}$</td>
<td>90</td>
<td>Used to refer to the old-mode during any mode transition.</td>
</tr>
<tr>
<td>$M^{new}$</td>
<td>90</td>
<td>Used to refer to the new-mode during any mode transition.</td>
</tr>
<tr>
<td>$\text{sched}(\pi, S, \tau^f)$</td>
<td>93</td>
<td>The function used by AM-MSO in order to determine whether a task $\tau^f$ can be safely enabled and then scheduled by $S$ on the platform $\pi$.</td>
</tr>
<tr>
<td>$\text{idle}_k(J, \pi, P)$</td>
<td>94</td>
<td>The $k^{th}$ idle-instant in the schedule of the job set $J$ upon the platform $\pi$ according to the job priority assignment $P$.</td>
</tr>
<tr>
<td>$\overline{\text{idle}}_k(J, \pi, P)$</td>
<td>94</td>
<td>An upper-bound on the $k^{th}$ idle-instant $\text{idle}_k(J, \pi, P)$.</td>
</tr>
<tr>
<td>Symbol</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>$\text{idle}_k(J, \pi)$</td>
<td>98</td>
<td>An upper-bound on $\text{idle}_k(J, \pi, X)$, for all job priority assignments $X$.</td>
</tr>
<tr>
<td>$\text{ms}^{\text{ident}}(J, \pi)$</td>
<td>124</td>
<td>An upper-bound on the makespan for the set of job $J$ and the identical multiprocessor platform $\pi$.</td>
</tr>
<tr>
<td>$\text{Work}_k^i$</td>
<td>126</td>
<td>The processed work on processor $\pi_k$, considering only the $i$ highest priority jobs of $J$.</td>
</tr>
<tr>
<td>$\text{idle}_k(J, \pi)$</td>
<td>134</td>
<td>A lower-bound on $\text{idle}_k(J, \pi, X)$, for all job priority assignments $X$.</td>
</tr>
<tr>
<td>$\text{ms}^{\text{unif}}_1(J, \pi)$</td>
<td>137</td>
<td>Our first upper-bound on the makespan for the set $J$ of jobs and the uniform platform $\pi$.</td>
</tr>
<tr>
<td>$\text{ms}^{\text{unif}}_0(J, \pi)$</td>
<td>138</td>
<td>The naive extension of the upper-bound $\text{ms}^{\text{ident}}(J, \pi)$ to uniform platforms.</td>
</tr>
<tr>
<td>$\text{idle}_j(J_i, \pi, \mathcal{P})$</td>
<td>140</td>
<td>The $k$th idle-instant in the schedule by $\mathcal{P}$ upon $\pi$ of the jobs with a higher priority than $J_i$ according to $\mathcal{P}$.</td>
</tr>
<tr>
<td>$\text{comp}_j(J_i, \pi, \mathcal{P})$</td>
<td>140</td>
<td>The $k$th idle-instant in the schedule by $\mathcal{P}$ upon $\pi$ of the jobs with a higher (or equal) priority than $J_i$ according to $\mathcal{P}$.</td>
</tr>
<tr>
<td>$\text{comp}_1(J_i, \pi, \mathcal{P})$</td>
<td>140</td>
<td>An upper-bound on the completion time of job $J_i \in J$ in the schedule of the job set $J$ upon the platform $\pi$ according to the job priority assignment $\mathcal{P}$.</td>
</tr>
<tr>
<td>$\text{ms}^{\text{unif}}_2(J, \pi)$</td>
<td>145</td>
<td>Our second upper-bound on the makespan for the set $J$ of jobs and the uniform platform $\pi$.</td>
</tr>
<tr>
<td>$s_x$</td>
<td>163</td>
<td>A critical speed.</td>
</tr>
</tbody>
</table>
### Table of symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\overline{m}_{3}^{\text{unif}}(J, \pi)$</td>
<td>166</td>
<td>Our third upper-bound on the makespan for the set $J$ of jobs and the uniform platform $\pi$.</td>
</tr>
<tr>
<td>$\lambda_{\pi}$</td>
<td>187</td>
<td>This parameter measures the “degree” by which $\pi$ differs from an identical multiprocessor platform, i.e., its “degree of heterogeneity”.</td>
</tr>
</tbody>
</table>

### Chapitre 3: Optimizing the hardware design

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMP</td>
<td>204</td>
<td>Our dual CPU type Multiprocessor System-on-Chip (MPSoC) platform.</td>
</tr>
<tr>
<td>$L_{lp}$</td>
<td>204</td>
<td>The platform of the DMP only composed of low-power and low-performance processors.</td>
</tr>
<tr>
<td>$m_{lp}$</td>
<td>204</td>
<td>The number of processors in $L_{lp}$.</td>
</tr>
<tr>
<td>$L_{hp}$</td>
<td>204</td>
<td>The platform of the DMP only composed of high-power and high-performance processors.</td>
</tr>
<tr>
<td>$m_{hp}$</td>
<td>204</td>
<td>The number of processors in $L_{hp}$.</td>
</tr>
<tr>
<td>$x$</td>
<td>206</td>
<td>Any of the two notations $lp$ and $hp$. For instance, $L_{x}$ denotes any of the two platforms of the DMP, i.e., $L_{lp}$ or $L_{hp}$</td>
</tr>
<tr>
<td>$\text{Pwr}_{\text{run}}^{x}$</td>
<td>206</td>
<td>The estimated power dissipation of an $x$-processor while it is running a task at its maximal clock frequency.</td>
</tr>
<tr>
<td>$\text{Pwr}_{\text{idle}}^{x}$</td>
<td>206</td>
<td>The estimated power dissipation of an $x$-processor while it idles.</td>
</tr>
<tr>
<td>$C_{lp}^{\tau_{i}}$, $C_{hp}^{\tau_{i}}$</td>
<td>206</td>
<td>The worst-case execution time of the task $\tau_{i}$ when executed on platform $L_{lp}$ and $L_{hp}$, respectively.</td>
</tr>
<tr>
<td>$U_{i}^{x}$</td>
<td>207</td>
<td>The utilization of the task $\tau_{i}$ on platform $L_{x}$.</td>
</tr>
</tbody>
</table>
### Table of symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_{\text{sum}}^x(\tau)$</td>
<td>207</td>
<td>The generalized utilization of the set of tasks $\tau$ on platform $L_x$.</td>
</tr>
<tr>
<td>$U_{\text{max}}^x(\tau)$</td>
<td>207</td>
<td>The maximal utilization of the set of tasks $\tau$ on platform $L_x$.</td>
</tr>
<tr>
<td>$\delta_i^x$</td>
<td>208</td>
<td>The density of the task $\tau_i$ on platform $L_x$.</td>
</tr>
<tr>
<td>$\delta_{\text{sum}}^x(\tau)$</td>
<td>208</td>
<td>The generalized density of the set of tasks $\tau$ on platform $L_x$.</td>
</tr>
<tr>
<td>$\delta_{\text{max}}^x(\tau)$</td>
<td>208</td>
<td>The maximal density of the set of tasks $\tau$ on platform $L_x$.</td>
</tr>
<tr>
<td>DBF$^x(\tau_i, t)$</td>
<td>208</td>
<td>The demand bound function of the task $\tau_i$ on platform $L_x$ in a time interval of length $t$.</td>
</tr>
<tr>
<td>LOAD$^x(\tau)$</td>
<td>209</td>
<td>The load of the task set $\tau$ on platform $L_x$.</td>
</tr>
<tr>
<td>$\tau^\text{lp}, \tau^\text{hp}$</td>
<td>211</td>
<td>The subset of $\tau$ which is exclusively executed on platform $L_{\text{lp}}$ and $L_{\text{hp}}$, respectively.</td>
</tr>
<tr>
<td>$S^\text{lp}, S^\text{hp}$</td>
<td>214</td>
<td>The scheduling algorithm used on platform $L_{\text{lp}}$ and $L_{\text{hp}}$, respectively.</td>
</tr>
<tr>
<td>CPU$^x(S)(\tau)$</td>
<td>214</td>
<td>The function that returns a sufficient number of $x$-processors so that the task set $\tau$ can be scheduled by $S^x$ on platform $L_x$ without missing any deadline.</td>
</tr>
<tr>
<td>Pwr$^x(\tau)$</td>
<td>218</td>
<td>An upper-bound on the power dissipation of the platform $L_x$ executing the task set $\tau$.</td>
</tr>
<tr>
<td>Pwr($\tau^\text{lp}, \tau^\text{hp}$)</td>
<td>219</td>
<td>An upper-bound on the power dissipation of a DMP.</td>
</tr>
<tr>
<td>$m^\text{on}<em>{\text{lp}}(M^i), m^\text{on}</em>{\text{hp}}(M^i)$</td>
<td>230</td>
<td>The number of processors powered on in the platforms $L_{\text{lp}}$ and $L_{\text{hp}}$ (respectively) during the execution of the mode $M^i$.</td>
</tr>
</tbody>
</table>
### Table of symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_{lp}(M^i), \tau_{hp}(M^i)$</td>
<td>The set of tasks executed on the platforms $L_{lp}$ and $L_{hp}$ (respectively) during the execution of the mode $M^i$.</td>
<td>230</td>
</tr>
<tr>
<td>$m_{lp}^{on}(M^i, M^j), m_{hp}^{on}(M^i, M^j)$</td>
<td>The number of processors powered on in the platforms $L_{lp}$ and $L_{hp}$ (respectively) during every transition phase from mode $M^i$ to mode $M^j$.</td>
<td>230</td>
</tr>
</tbody>
</table>

---

### Chapitre 4: Exploiting the DVFS framework

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K$</td>
<td>The number of available levels of $&lt;$voltage, frequency$&gt;$.</td>
<td>247</td>
</tr>
<tr>
<td>$s^1, s^2, \ldots, s^K$</td>
<td>The speeds available to the processors.</td>
<td>247</td>
</tr>
<tr>
<td>$s_{min} = s^1, s_{max} = s^K$</td>
<td>The minimal and maximal available speed of the processors, respectively.</td>
<td>247</td>
</tr>
<tr>
<td>$Pwr_{run}(s^i)$</td>
<td>The relative power dissipation of a processor while it is running at speed $s^i$.</td>
<td>247</td>
</tr>
<tr>
<td>$sched(S, \tau, m, s)$</td>
<td>The function used by our identical speed determination process in order to determine whether a task set $\tau$ is schedulable by $S$ on the DVFS-identical platform $\pi$ in which all the processors are running at speed $s$.</td>
<td>264</td>
</tr>
<tr>
<td>$sched(\pi, S, \tau)$</td>
<td>The function used by our individual speed determination process in order to determine whether a task set $\tau$ is schedulable by $S$ on the uniform platform $\pi$.</td>
<td>275</td>
</tr>
</tbody>
</table>
### Table of symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>worst-case schedule</td>
<td>277</td>
<td>The schedule in which every job executes for its WCET.</td>
</tr>
<tr>
<td>actual schedule</td>
<td>277</td>
<td>The schedule which is actually produced at run-time.</td>
</tr>
<tr>
<td>$s_i$</td>
<td>282</td>
<td>The execution speed of the active job of $\tau_i$.</td>
</tr>
<tr>
<td>$s_i^{\text{off}}$</td>
<td>282</td>
<td>The offline speed of every job of $\tau_i$.</td>
</tr>
<tr>
<td>$e_i$</td>
<td>283</td>
<td>This parameter reflects the difference between the consumption profile of task $\tau_i$ and the power dissipation measured while running benchmarks.</td>
</tr>
<tr>
<td>$E_i(R, s)$</td>
<td>283</td>
<td>The amount of energy consumed by the task $\tau_i$ when executed for $R$ time units at speed $s$.</td>
</tr>
<tr>
<td>offline schedule</td>
<td>284</td>
<td>The schedule in which every job $\tau_{i,j}$ runs at its offline speed $s_i^{\text{off}}$ and executes for its WCET $C_i$.</td>
</tr>
<tr>
<td>$\text{rem}_i(t)$</td>
<td>284</td>
<td>The remaining worst-case execution time of the active job $\tau_{i,j}$ of $\tau_i$ in the actual schedule if $\tau_{i,j}$ completes its execution at speed $s^{\text{max}}$.</td>
</tr>
<tr>
<td>$\text{rem}_i^{\text{off}}(t)$</td>
<td>284</td>
<td>The remaining worst-case execution time of the active job $\tau_{i,j}$ of $\tau_i$ in the offline schedule if $\tau_{i,j}$ completes its execution at speed $s^{\text{max}}$.</td>
</tr>
<tr>
<td>$\text{disp}_j^{\text{off}}(t)$</td>
<td>284</td>
<td>The earliest instant after time $t$ such that $\tau_{i,j}$ is dispatched in the offline schedule, considering only the set of jobs that are active at time $t$ in the offline schedule.</td>
</tr>
<tr>
<td>$\text{nextdisp}_j^{\text{off}}(\pi_k, t)$</td>
<td>284</td>
<td>The earliest instant after time $t$ at which a job which is active in the actual schedule at time $t$ is dispatched to $\pi_k$ in the offline schedule, considering only the set of jobs that are active at time $t$ in the offline schedule.</td>
</tr>
<tr>
<td>Symbol</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LastRel(_i(t))</td>
<td>299</td>
<td>The latest instant before time (t) at which the task (\tau_i) released a job.</td>
</tr>
<tr>
<td>PotRel(_i(t, t'))</td>
<td>299</td>
<td>The potential release function indicates whether the task (\tau_i) could release a job at time (t' &gt; t), where (t) is the current time.</td>
</tr>
<tr>
<td>rem(_i(t))</td>
<td>300</td>
<td>The remaining worst-case execution time of the last released job of task (\tau_i) if executed at maximal processor speed (s_{\text{max}}).</td>
</tr>
<tr>
<td>PotAct(_i(t, t'))</td>
<td>300</td>
<td>The potentially active function PotAct(_i(t, t')) indicates whether (\tau_i) has an active job at the current time (t) that could be still active at a future time (t').</td>
</tr>
<tr>
<td>(\Pi_i(t, t'))</td>
<td>300</td>
<td>When non-negative, the function (\Pi_i(t, t')) provides a lower bound on the number of processors that will idle at time (t' \geq t) (where (t) is the current time), while ignoring the schedule of the active job (\tau_{i,j}) (if any).</td>
</tr>
<tr>
<td>(t_{\text{next}})</td>
<td>301</td>
<td>Whenever any job (\tau_{i,j}) is dispatched to any CPU (\pi_k) at time (t), (t_{\text{next}}) is the earliest future instant in the schedule at which another job (possibly from the same task) could have no other choice than to be dispatched to (\pi_k).</td>
</tr>
</tbody>
</table>