A semi-partitioned approach for parallel real-time scheduling

Benjamin Bado
Université Libre de Bruxelles
benjamin.bado@ulb.ac.be

Laurent George
ECE Paris - LACSC
laurent.george@ece.fr

Pierre Courbin
ECE Paris - LACSC
pierre.courbin@ece.fr

Joël Goossens
Université Libre de Bruxelles
joel.goossens@ulb.ac.be

ABSTRACT
In this paper, we consider the problem of scheduling periodic Multi-Phase Multi-Thread tasks on a set of m identical processors with Earliest Deadline First (EDF) scheduling. Each periodic task is defined by a sequence of phases with offsets that can be possibly parallelized. We use a portioned semi-partitioned approach with migrations at local deadlines assigned to each phase. We extend this approach to take into account phase parallelism. The phase parallelism we consider is an extension of the popular job parallelism. A phase, if parallelizable, can be decomposed into parallel threads run on a configurable number of processors. We only require simultaneous execution of threads inside a window equal to the local deadline of their associated phase. To decide on the schedulability of a Multi-Phase Multi-Thread task, we extend the popular uniprocessor EDF feasibility condition for periodic asynchronous tasks. We propose two new schedulability tests for EDF that significantly improve the well known Leung and Merrill feasibility test based on the feasibility interval $[O_{\text{min}}, O_{\text{max}} + 2P]$, where $O_{\text{min}}$ and $O_{\text{max}}$ are respectively the minimum and maximum phase offsets and $P$ the least common multiple of the task periods. The first schedulability test is used when an EDF simulation is needed and gives, by simulation, a 44% gain in simulation speed. The second method provides a sufficient schedulability test with a time interval of length $P$ based on the demand bound function. Finally, we study three local deadline assignment heuristics assigned to parallelizable phases. We compare and analyze the performances obtained by simulation for those three local deadline assignment heuristics.

Keywords
real-time scheduling theory

Categories and Subject Descriptors
C.3 [Real-time and embedded systems]:

General Terms
Theory

1. INTRODUCTION
The study of parallel multiprocessor real-time systems is quite recent, and few results have yet been found. However there exist different models that represent this problem. Parallelism adds the possibility for some specific parts of the tasks to be executed at the same time on different processors. In this article we focus on the problem of multiprocessor parallel real-time systems for tasks following a periodic Multi-Phase Multi-Thread task model (see Section 2.1).

Each task is represented as a sequence of parallel and sequential executions. After a sequential execution, a fork is done, which means splitting the phase into threads that are executed on a set of processors. After a parallel execution, a join is made to synchronize and end the parallel threads.

There are three different approaches for scheduling tasks on multiprocessor system: global scheduling, partitioned scheduling and semi-partitioned scheduling.

Global scheduling. This approach aims to find a global strategy for scheduling jobs (task instances), which are allowed to migrate between processors.

Partitioned scheduling. This approach aims to find a strategy for partitioning tasks on processors; once a task is assigned to a processor, it cannot migrate. Once the distribution of the tasks has been done, a uniprocessor scheduling feasibility condition is used to decide on the schedulability of the system.

Semi-Partitioned scheduling. This approach aims to assign tasks in such a way that the migrations of a task follow a migration pattern. There are two approaches:

- One where a job can be executed on only one processor, but an other job of the same task can be executed on different processors. This approach is also referred to as the restricted migration case [4].

- One where the Worst Case Execution Time (WCET) of a job is split into different parts that can be assigned to different processors. This approach is known as job portion migration [9]. In this paper, we focus on this approach.
Besides there are three categories of parallel tasks: (1) rigid: the number of processors simultaneously assigned to a task is fixed a priori and cannot change over time, (2) moldable: the number of processors assigned simultaneously to a task can vary, but it does not vary with time for a given job, (3) malleable: the number of processors assigned simultaneously to a job can vary. In this paper we focus on Semi-Partitioned scheduling with moldable parallel tasks.

1.1 Our contributions

Parallel execution means, in the state-of-the-art, simultaneous executions of threads. In this paper, we extend the parallelism of a job by requiring that all parallel threads associated to a phase must be executed in the same time interval but not necessarily at the same time for all their durations. In our model, we split tasks into phases in a way that the phase of a task cannot begin before its previous phase. This is done by assigning offsets and local deadlines to each phase. This construction cancels the phenomenon of jitter between threads. For parallel threads, this leads to releasing all threads at the same time.

We present a two-steps method to schedule parallelizable phases with a semi-partitioned approach and provide three algorithms to assign local deadlines to phases. We test the efficiency of each of the three algorithms and we compare them by simulation. One of this algorithms uses EDF uniprocessor simulations to assign local deadlines and local offsets to phases. For those kinds of tasks, Leung et al. [11] proved that a schedulability test consists in verifying that all tasks meet their deadlines when scheduled EDF in the time interval \([O_{\text{min}}, O_{\text{max}} + 2P]\), where \(O_{\text{min}}\) and \(O_{\text{max}}\) are respectively the minimum and maximum phase offsets and \(P\) the least common multiple of the task periods.

We provide new schedulability results for EDF uniprocessor scheduling of periodic asynchronous tasks. As in [2], we improve the popular study interval \([O_{\text{min}}, O_{\text{max}} + 2P]\) [11] with two new schedulability tests. For the first one, we extend the result of [2] by proving that the feasibility interval of EDF can be restricted to the first idle time after \(O_{\text{max}} + P\) (if any) or to \(O_{\text{max}} + 2P\) otherwise.

The second method provides a sufficient feasibility condition for periodic asynchronous tasks scheduled by EDF based on a modification of the demand bound function. We show that we must only consider jobs having release times and absolute deadlines in the time interval \([O_{\text{max}} + P, O_{\text{max}} + 2P]\) w.r.t to the corresponding scenario with offsets.

1.2 Organization of the paper

The remainder of this paper is organized as follows. Section 2 describes our model and introduces some definitions. Section 3 presents a state of the art for parallel scheduling. Section 4 presents our semi-partitioned approach. We then focus on the local deadline assignment problem in Section 5 where we present three different methods to allocate local deadlines to phases. Section 6 presents new results on uniprocessor asynchronous systems. Section 7 presents the simulation results and performance evaluations of the algorithm defined in Section 6. Finally, we conclude in Section 8.

2. CONCEPTS AND NOTATIONS

In this work we consider a platform of \(m\) unit-capacity identical processors. In the next subsections we present our

![Figure 1: Representation of a Multi-Phase Multi-Thread Task](image-url)
• $s_i^1 = 0$, i.e., the arrival time of the first phase of the task corresponds to the arrival time of the task itself.

• $\forall j > 1, s_i^j = s_i^{j-1} + f_i^{j-1}$, i.e., the relative arrival offset of a phase is larger than the deadline of the previous phase. In other words, we solve the precedence constraint between successive phases using relative arrival offsets and local deadlines.

• $s_i^\text{f} + f_i^\text{f} = D_i$, i.e., the deadline of the last phase corresponds to the deadline of $\tau_i$.

• All tasks are independent of each other. This means that no mutual exclusion can exist between two tasks.

• We follow the ideas of the Fork-Join task model since:
  – we impose a sequence of sequential and parallel phases. Each phase $\phi_j^i$ with $j$ an odd number will be a sequential phase.
  – we impose that each task has to start and finish with a sequential phase. Note that $\ell_j^i$ is then an odd number.

2.2 Definitions for our model

• We will say that a phase cannot be parallelized if the parallelization cost $\gamma_j^j$ is equal to $\infty$ for all values of the vector.

• $C_i^j \overset{\text{def}}{=} \sum_{j=1}^{\ell_j^i} C_i^j$ is the total WCET of the task $\tau_i$.

• $C_i,\text{Par} \overset{\text{def}}{=} \sum_{j=1}^{\ell_j^i} C_i^j, j \neq \ell_j^i$ is the WCET part of the task $\tau_i$ which can be parallelized (set to 0 if the task cannot be parallelized).

• $u_i \overset{\text{def}}{=} \frac{C_i^1}{\lambda_i}$ is the utilization of the task $\tau_i$.

• $U \overset{\text{def}}{=} \sum_{i=1}^n u_i$ is the total system utilization.

• $\lambda_i \overset{\text{def}}{=} \frac{C_i}{\min(T_i, D_i)}$ is the density of the task $\tau_i$.

• $\lambda_i,\text{Par} \overset{\text{def}}{=} \frac{C_i,\text{Par}}{\min(T_i, D_i)}$ is the density of parallel phases of the task $\tau_i$.

**Definition 1. Phase parallelism** A phase $\phi_j^i$ is executed in parallel if it is executed on more than one processor and if all its threads are executed in the same time interval $[t + s_i^j, t + f_i^j]$ where $t$ represents the time when a job of the task $\tau_i$ is released.

This definition of parallelism is weaker than the popular definition of parallelism which requires that all threads are executed at the same time for all their durations (i.e. the Gang Model). In our model, the phases have to be executed in the same time interval (not necessarily at the same time).

2.3 Definitions for uniprocessor systems

• A feasibility interval for EDF [11] with $U \leq 1$ is $[0, O_{\text{max}} + 2P]$ where $O_{\text{max}} \overset{\text{def}}{=} \max\{O_1, \ldots, O_n\}$ and $P$ denotes the least common multiple (LCM) of all the periods in task set $\tau$: $P \overset{\text{def}}{=} \text{lcm}\{T_1, \ldots, T_n\}$.

• An idle time $t$ is a time such that all requests released strictly before $t$ have completed their execution before or at time $t$.

• DBF*(t) (Demand Bound Function) is the amount of processing time required by all tasks, whose release times and absolute deadlines are in the time interval $[0, t]$ in a given offset scenario.

3. STATE OF THE ART

• Han et al. [8] have proved the NP-Completeness for the problem of assigning fixed priority to parallel tasks. An heuristic algorithm is also proposed, where the number of processors assigned to the job is chosen by the scheduler.

• Manimaran et al. [14] defined a way to improve the benefits of task parallelism by an offline scheduling algorithm followed by a run-time scheduling algorithm with tasks that can be added dynamically to the system. A task can be split into parallel subtasks(phases) that have to start at the same time. The offline scheduler is a non-preemptive EDF algorithm where the minimum parallelism is used when a deadline cannot be met. The run-time scheduler plays on the degree of parallelism of a task to avoid run-time anomaly (which may cause some of the schedulable tasks sets to be non schedulable). By simulation, the algorithm is shown to be more efficient than the non-parallel version of non-preemptive EDF algorithm.

• Collette et al. [3] use a malleable tasks assignment model and introduce the notion of work-limited parallelism. A cost is assigned to the parallelism, so the gain in performance of a task being executed on $j+1$ processors instead of $j$ will be higher than or equal to the gain in performance of a task being executed on $j+2$ instead of $j$. They propose a scheduling algorithm using the notion of a minimal required number of processors needed to execute a job and provide an utilization bound for this kind of system.

• Lakshmananan et al. [10] use a moldable task assignment model on a basic Fork-Join task model, where a task must begin and end with a single sequential thread (no parallel part at the beginning and at the end of a task). They found a worst task set scenario for a fork-join structure such that the system is not schedulable with the smaller utilization, and they also found a best task set scenario such that the system is feasible with the highest utilization. The article give also a transformation model to the tasks to minimize the Fork-Join structure as much as possible by using a stretch transformation. Finally a deadline-monotonic partitioning algorithm is used to provide a resource augmentation bound of 3.42, which means that any task set that is feasible on $m$ unit speed processors can be scheduled by their algorithm on $m$ processors that are 3.42 times faster.

• Berten et al. [1] introduce a specific multi-thread parallel task model and provide a necessary schedulability test and integrate precedence constraints in their calculation.
• Saifullah et al. [15] proposed a model where a task can contain many segments, that can run on an arbitrary number of processors. Playing on the slack (the slack is defined as the difference between the deadline and the WCET of the task) of a task to find local deadlines and offsets, a decomposition of task into a set of phases (corresponding to the different segments of the task) is proposed. The authors prove a resource augmentation bound of 2.62 in a global scheduling and of 3.42 in a partitioned scheduling when using this decomposition method. In this paper, a DAG model is also analyzed.

To the best of our knowledge, no study has been carried out on semi-partitioned approaches applied to parallel tasks.

4. SCHEDULING

The semi-partitioned scheduler will work in two steps. First it will allocate as many tasks as possible to the processors using a partitioned approach (step S1). After the first step, the scheduler will use another algorithm to split the tasks into phases assigned to a subset of processors (step S2). In doing so, our approach dominates the partitioned approach. All the results of the partitioned approach can be reused for step S1.

4.1 The first step ‘S1’

The goal of this step is to use a partitioned approach to allocate as many tasks as possible to the processors. The aim is to avoid job migration as much as possible and use parallelism when the partitioned approach fails to assign a task. We use the Worst-Fit heuristic [12, 7] for this step: tasks are allocated sequentially in such a way that the remaining processor utilization is maximized. At the end of this heuristic, the remaining tasks that could not be assigned to a single processor, will reduce their WCET by parallelizing their phases. To improve the opportunities for parallelization in the second step, we will assign the less interesting tasks in term of parallelism in the first step. For this, we will classify the tasks in three levels.

L1 This level is composed of the tasks that cannot be parallelized, which means that they have only one phase, composed of one non parallelizable thread. These tasks are sorted by decreasing values of their density, as this offers the best performance according to [13].

L2 This level is composed of the tasks that respect the equation: \(C_i = \sum_{j=1}^{k_i} C_i^j \leq D_i\). These are the tasks that can be scheduled on one processor. At this level we will sort the tasks by increasing value of \(C_i,Par\). In this way, the tasks that cannot be scheduled in the first step, will be those that are the most interesting to parallelize.

L3 This level is composed of the tasks that respect the equation: \(C_i = \sum_{j=1}^{k_i} C_i^j > D_i\). These are the tasks that must be parallelized in order to respect their deadline, i.e. the partitioning algorithm cannot assign a task from L3 to only one processor. Knowing that the goal of step S1 is to assign as many tasks as possible before the second step, the tasks of L3 must be assigned last in step S1. These tasks will be considered according to their need for parallelism. The need in parallelism is the density of a sequential task \(\tau_i\) that needs to be parallelized so that the sequential part of \(\tau_i\) can be scheduled on one processor. Mathematically this need is represented by \(C_{\text{ini}} \geq D_i\). If this value is negative or equal to zero, that means that the task can be scheduled on one processor without being parallelized.

The goal of this step is to assign as many tasks as possible with a partitioned approach, and keeping the tasks that are the most interesting to parallelize for step S2. That’s why level L1 will be the first to be assigned with the Worst-Fit heuristic, level L2 the second, and level L3 the third. Once there are no more tasks that can be entirely assigned to one processor, the algorithm will enter step S2.

4.2 The second step ‘S2’

In this step we allocate the remaining tasks, i.e. tasks which are not allocated in the first step. If there are still some tasks remaining after level L1, it means that there are still some tasks that cannot be parallelized, we will allocate these tasks using the portioned semi-partitioned scheduling method proposed by George et al. [5]. In practice, if a job cannot be executed on a single processor, it is portioned and different parts of the task are executed on different processors in a way that the number of migrations is limited. We use their method for the tasks defined in their article (tasks without parallelism) and we introduce new methods that use the benefits of parallelism for the tasks that can be parallelized. For the other tasks, we first try to assign those that may need the most processors, i.e. those of level L3 before those of level L2. We also begin by sorting the processors by decreasing value of density.

Remark.

Knowing that the cost of a Fork-Join operation, of a migration between processor, and the cost of parallelism, we will try in this method to parallelize tasks as less as possible. In our method, when a parallelizable task \(\tau_i\) cannot be scheduled on one processor \(P\), we will decrease the WCET of \(\tau_i\) that will be assigned to \(P\) by parallelizing its phases. To ensure that the threads of the different phases will executes in the same time interval, we must define local offsets and local deadlines to each phase. For this, we have defined 3 methods of assigning local deadlines that will be described in next Section. In this method, we also use the notion of task \(\tau_i^{temp}\) that is defined as follows :

Definition 2. The task \(\tau_i^{temp}\) of a task \(\tau_i\) represents the biggest part of the task \(\tau_i\) that will be assigned to a single processor, and is constructed by taking the first threads of each phase of \(\tau_i\).

Figure 2 illustrates the concept of task \(\tau_i^{temp}\). The part of algorithm representing step 2 is defined in Algorithm 1.

Remark.

The method to assign the threads of a task \(\tau_i\), that are not in \(\tau_i^{temp}\) (in Algorithm 1 line 16 and 18) just have to respect the following implicit rules: two threads of a same phase can not be assigned to a same processor.

Task Transformation

An important choice has to be made in the way of splitting threads and assigning them local deadlines and offsets. Since
In the following, we propose three local deadline assignment methods in section 7. We then compare the efficiency of those methods that take the remaining tasks and try to assign them sequentially. We can reduce this problem to just finding local deadlines.

Algorithm 1: Step 2

Require: Remaining tasks of Step 1
1: assign the tasks of L1 with the method defined in [5]
2: for Each task \( \tau_i \) taken sequentially and \( \tau_i \) can still be parallelized do
3: for Each parallelizable phase \( \phi_j \) of task \( \tau_i \) do
4: Compute the gain in WCET of the phase. A phase parallelized on \( k \) processors has a gain in WCET equal to \( \left( \frac{C_{\phi_j}}{k} \right) \left( 1 + \gamma_{\phi_j}^k \right) - \left( \frac{C_{\phi_j}}{k+1} \right) \left( 1 + \gamma_{\phi_j}^{k+1} \right) \)
5: (a negative gain means that the task can no more be parallelized)
6: end for
7: Let \( j \) be the identifier of the phase with the biggest gain in WCET.
8: if the gain in WCET of \( \phi_j^i \) is a negative value then
9: The task can no more be parallelized, the system is thus not schedulable
10: else
11: assign a new processor to phase \( \phi_j^i \).
12: Update the thread of phase \( \phi_j^i \) with the new created thread and with the new cost of parallelism.
13: Create task \( \tau_i^{temp} \) of \( \tau_i \).
14: Use a method of local deadline and local offset allocation on \( \tau_i^{temp} \).
15: if The task \( \tau_i^{temp} \) can be allocated to one processor \( x \) then
16: if the threads of \( \tau_i \) that are not in \( \tau_i^{temp} \) can be assigned to the other processors then
17: assign the threads of \( \tau_i \) that are in \( \tau_i^{temp} \) to processor \( x \)
18: assign the remaining threads of \( \tau_i \) to other processors.
19: return true // the system is schedulable
20: end if
21: end if
22: end if
23: end for
24: return false // the system is not schedulable

5. “LOCAL DEADLINES” ASSIGNMENT METHODS

In this part we propose three local deadline assignments to phases, and analyze their benefits and limitations.

Goal. Knowing that the costs of parallelism and migrations are non negligible, we will try to limit the parallelism to the minimum required. The idea of minimizing the parallelism was used in a Stretch transformation algorithm [10]. Our strategy takes the tasks that were not scheduled sequentially during step 1. We try to assign these tasks in such a way that we assign to one processor the longest part of WCET as possible.

As reminder, the following three methods work for tasks \( \tau_i^{temp} \) that means tasks composed of a set of phases with only one thread.

5.1 Local Fair Deadline assignment

This allocation gives the same deadline to each phase equal to the ratio \( \frac{C_j}{p} \times D_i \). The part of \( D_i \) that was not assigned is given to the phase that is the most parallelized (the biggest \( v_j^i \)) to minimize the utilization on the biggest number of processors.

5.2 Minimum Local Deadline assignment

This strategy aims to assign each phase \( \phi_j^i \) of a sequential task \( \tau_i \) in a sequential way on a processor \( p \). In practice:

- The offset of \( \phi_j^i \) will be equal to the offset of \( \tau_i \). The initial value of \( f_j^i = C_{\phi_j}^i \)
- Try to assign \( \phi_j^i \) to processor \( p \).
- While the phase can not be scheduled on processor \( p \), increment the value of \( f_j^i \)
- If the value of \( f_j^i \) is bigger than \( D_i - C_{\phi_j}^i + C_{\phi_j}^i \), that means that the other phases could not be scheduled. The part of \( D_i \) that can still be given to phases that were not yet assigned to processor \( p \) is lower than the sum of \( C_{\phi_j}^i \) of those phases. That means that \( \tau_i \) is not schedulable on processor \( p \).
- If \( \phi_j^i \) was assigned, try the same schema with the other phases.

5.3 Local Deadline set to the worst case response time

THEOREM 1. When a task set is schedulable with EDF on the study interval, we can assign to the local deadlines of the phases of a task, the worst response time that was experienced by this phase during the simulation.

Proof. There are two results to demonstrate: (1) with the new deadlines, the task won’t miss any deadline and (2) the other tasks won’t miss any deadline.

(1) trivial, knowing that each phase will have a deadline lower than or equal to the deadline of the task, it will have at least the same priority, so it will also be scheduled.
This assignment is done by respecting the rule defined in theorem 1. So we need to first schedule the task \( \tau_i \) with EDF, considering all the phases of \( \tau_i \) as one single thread. We have to remember the relative worst execution of each phase during the simulation (looking at the time when the single thread executed a certain amount of \( C_j^i \) ) and assign this value as \( f_j^i \) to the phases. The pseudo-code of this algorithm is presented in Algorithm 2.

6. Scheduling Conditions

The “local deadline” assignment methods can use a uniprocessor EDF scheduling test many times, so we need to improve these tests to accelerate the computation. We will use the Demand Bound Function (DBF) test rather than a simple simulation to save computation time.

In a synchronous scenario, \( \text{DBF}(t) \) is defined as the amount of processing time required by all tasks, whose release times and absolute deadlines are in the time interval \([0, t]\). \( \text{DBF}(t) = \sum_{i=1}^{n} \text{DBF}_i(t)/C_i \) where \( \text{DBF}_i(t) = \max(0, 1 + \left\lfloor \frac{t - O_i}{T_j} \right\rfloor) \). An existing scheduling condition for EDF using the notion \( \text{DBF}(t) \) was proposed by George et al. [6]. This scheduling condition is defined as follow:

\[
\max \left\{ \sum_{i \in M} \frac{h_i(t)}{t} \right\} \leq 1 \text{ where } M = \bigcup_{j=1}^{n} \left\{ D_j + k_jT_j, 0 \leq k_j \leq \left\lfloor \frac{P - D_j}{T_j} \right\rfloor - 1 \right\}
\]

This condition needs to be extended to asynchronous scenarios.

Definition 3. \( \text{DBF}^*_i(t) = \max \left\{ 0, 1 + \left\lfloor \frac{t - O_i}{T_j} \right\rfloor \right\} \), and where \( \text{DBF}^*_i(t) \) is the offset extension of \( \text{DBF}(t) \).

With asynchronous scenarios, we need to find a feasibility interval for \( \text{DBF}^*_i(t) \) and make some modifications on the utilization condition.

6.1 Study interval for \( \text{DBF}^*_i(t) \)

Choquet-Geniet et al. [2] showed that a task system with offset has permanent phase, with a period \( P \), that begins in the interval \([O_{\text{max}}, O_{\text{max}} + P]\). They also proposed an algorithm to find when this permanent cycle begins. They defined a Request\((t)\) function.

Definition 4. Request\((t)\) is a function that represents the sum of the remaining computing times of the pending tasks at time \( t \).

We use one of their properties and analyze the schedulability problem in another way than that defined by Choquet-Geniet et al. Instead of looking for the first time when the cycle of size \( P \) appears, we will look for the time when this cycle ends. Our approach provides more efficient results in practice.

Algorithm 2: EDF local deadline assignment

Require: \( \tau_i \) list of processors
Ensure: a boolean saying if \( \tau_i \) was assigned. If the algorithm returns true, the algorithm has also assigned \( \tau_i^{\text{temp}} \) on the different processors.
1: isAssigned=false
2: while \( \tau_i \) can still be parallelized and isAssigned=false do
3: for Each parallelizable phase \( \phi_j^i \) of task \( \tau_i \) do
4: Compute the gain in WCET of the phase. A phase parallelized on \( k \) processors has a gain in WCET equal to \( \frac{C_i}{k} \left( 1 + \gamma^j \right) - \frac{C_i}{k+1} \left( 1 + \gamma^j \right) \)
5: (a negative gain means that the task can no more be parallelized)
6: end for
7: Let \( j \) be the identifier of the phase with the biggest gain in WCET.
8: if the gain in WCET of \( \phi_j^i \) is a negative value then
9: The task can no more be parallelized, the system is thus not schedulable
10: else
11: assign a new processor to phase \( \phi_j^i \).
12: Update the thread of phase \( \phi_j^i \) with the new created thread and with the new cost of parallelism.
13: Create task \( \phi_j^{\text{temp}} \) of \( \tau_i \).
14: if \( \phi_j^{\text{temp}} \) can be entirely assigned to a processor \( p \) with algorithm EDF (simulation test) then
15: \( s_k^i = O_i \)
16: \( f_j^i = \) worst response time of the equivalent phase in \( \phi_j^{\text{temp}} \) during the EDF simulation
17: for all \( \phi_k^i \) with \( k > 1 \) do
18: \( s_k^i = f_{j-1}^i \)
19: \( f_j^i = \) worst response time of the equivalent phase in \( \phi_j^{\text{temp}} \) during the EDF simulation
20: end for
21: assign the first thread of each phase \( \phi_j^i \) to processor \( p \)
22: isAssigned=true
23: end if
24: end if
25: end while
26: if isAssigned=false then
27: return false (not schedulable)
28: end if
29: Try to assign the remaining threads of \( \tau_i \) to other processors
30: If this assignment worked, return true (schedulable), else return false.

Property 1. In [2], Choquet-Geniet et al. proved that after \( O_{\text{max}} \), the cumulated processor request cannot decrease over a hyper period: \( \forall \Delta > 0, \text{Request}(O_{\text{max}} + \Delta + P) \geq \text{Request}(O_{\text{max}} + P) \).

Theorem 2. To test the schedulability of a set of Periodic tasks in an asynchronous scenario, we can stop the scheduling after the first idle time appearing after \( O_{\text{max}} + P \) if any and at time \( O_{\text{max}} + 2P \) otherwise.

Proof. Request\((t)\)=0 when an idle request appears, by definition of an idle time and of Request\((t)\). So if there is an
idle time at time $t$ after $O_{max} + P$, there is also an idle time at $t - P$. This is a direct consequence of property 1. By the property of $P$, we are in exactly the same situation at $t$ and $t - P$, i.e.: $\text{Request}(t) = \text{Request}(t - P) = 0$. To conclude, we are in the same state in $t$ and $t - P$, so the permanent period is between $t$ and $t - P$. The schedule can thus stop at $t$. □

**Consequence.** The worst case scheduling test ends at $O_{max} + 2P$, this case appearing when a task $\tau_i$ has the biggest offset, and has a period $T_i = P$, or if there are no idle time. When we schedule a system we do not need to compute a bound, we just need to stop after the first idle time after $O_{max} + P$. This point can be found with the method defined in Section 6.3. We have tested the efficiency of the algorithm by computing the ratio of the time at which our algorithm ends compared to the worst case $O_{max} + 2P$, i.e. $\frac{t}{O_{max} + 2P}$. For randomly generated task sets with a normal distribution, we found an average ratio close to 56% that can still reach 100% in the worst case, which means that on average there are few computations to do after time $O_{max} + P$.

**Theorem 3.** A schedulability interval for EDF uniprocessor of size $P$ is $[O_{max} + P, O_{max} + 2P]$.

**Proof.** By property 1, we know that the $\text{Request}(t)$ function after $O_{max} + P$ is higher than those at $t - P$, so if a deadline is missed before $O_{max} + P$ it will also be missed after because of a higher constraint. We know that there is a cycle of size $P$ that begins at the latest at time $O_{max} + P$, so by definition there is a cycle between $O_{max} + P$ and $O_{max} + 2P$, and this cycle is enough to analyze the schedulability of the system. □

**Definition 5.** The number of times where the processor was idle before time $t$ is represented by the difference between $t$ and the workload of all the tasks that are released at time $t$, this value is equal to $\zeta = \max 0, t - \sum_{i=0}^{n} \frac{t - O_i}{T_i}, C_i$. This is not an exact value: it is pessimistic. It count all the values of $C_i$ at each time. In a future work, we will remove this pessimism by finding an exact value for $\zeta$.

**Theorem 4.** A sufficient scheduling condition of size $P$, for an EDF uniprocessor on asynchronous tasks using $DBF^*(t)$ is $\max \big\{ \sum_{i=0}^{n} M \big\{ DBF^*(t)\}_{C_i} \big\} \leq 1$ where $M = [O_{max} + P, O_{max} + 2P]$.

**Proof.** We already proved that a study interval of size $P$ for EDF is $M = [O_{max} + P, O_{max} + 2P]$. Now we need to find a test that allows us to test a system immediately from a time $t$, without computing the system before $t$. This is done with the test on $DBF^*(t)$, $\sum_{i=0}^{n} M \big\{ DBF^*(t)\}_{C_i}$. By definition $DBF^*(t)$ is the amount of processing time required by all tasks, whose release times and absolute deadlines are in the time interval $[0, t]$. To have the real representation of the system at time $t$ we need to add to $DBF^*(t)$, all the times when the processor was idle $(\zeta)$ in the time interval $[0, t]$. The test is thus to see if the total requests of the tasks adding to the idle times of the system at time $t$, is lower than $\zeta$. That means that at time $t$, each task succeeds in receiving all its requests before its deadline. EDF being optimal, if the system is proved schedulable with our method, EDF can schedule this system. □

**Consequence.** With Theorem 4, there is no need to simulate a system to check its schedulability. Thanks to that we have a study interval of size $P$, which is much better than the actual of size $O_{max} + 2P$. The problem is due to the number of times when the processor was idle, which is too pessimistic in some cases leading to an only sufficient test.

**Comparison with results found by A. Choquet-Geniet [2].** A. Choquet-Geniet [2] proposed a method to find the first time when the periodic pattern begins, the problem is the computation of this value is quite long. Our first method that consists in stopping the schedule at the first idle time after $O_{max} + P$ has the advantage that there is no computation required to find when to stop the scheduling. For our second method, the scheduling condition of size $P$, our scheduling interval is of constant size, unlike theirs, which in the worst case can be of size $2P + O_{max}$. We also don’t need to do any computational heuristic before the scheduling, and in practice we analyze less than $P$ unit of time, because with the DBF we can just focus on the times when the deadline of a job appears.

### 6.2 Condition on the utilization $U$

With the constraint of offset, when we compute the utilization $U$, we can have cases where $U > 1$, and the system is still schedulable. This is because in the popular computation of $U$, we don’t pay attention to the fact that some tasks cannot be scheduled at the same time, because of the offset. In the following we will define this kind of task and explain how to solve the problem.

**Definition 6.** Two tasks $\tau_l$ and $\tau_j$ are said to have a transactional relation if they respect the three following conditions:

1. $T_i = T_j = T$
2. $O_i + D_i \leq O_j$
3. $O_i + D_i \leq O_j + T$

The first condition is a natural one that ensures that the tasks have the same period; the second condition ensures that the task $j$ cannot begin before the task $i$ has finished its execution, and the third condition ensures that a Job of $\tau_l$ must end before the beginning of a new Job of $\tau_j$.

**Remark 1.** The transactional relation is not transitive.

**Definition 7.** A group of transactional tasks is a task set that respects the following condition: each task of the set is transactional with all the other tasks of the set.

**Theorem 5.** When we consider all the tasks that are transactional as an unique task, constructed by summing their WCET and Deadline, and by taking the Period of one of the tasks considered (by definition of transactional tasks, they all have the same Period), the condition $U \leq 1$ becomes valid. This construction is only valid if we consider that each task belongs at most to one group of transactional tasks.

**Proof.** The proof is immediate knowing that the problem due to the computation of two tasks that can’t be executed at the same time is canceled. □
6.3 Method to find the first idle point after a time $t$

To mathematically find the time when to stop the simulation we can use the following method which looks for the first idle time after a time $t$, where here $t = O_{\text{max}} + P$.

- By definition of an idle time: the first idle time of the system, is the first time $t$ where the workload of all the tasks at this time is equal to $t$. Which means that all tasks are ended at this time.

- The number of times when the processor was idle before time $t$ is represented by the difference between $t$ and the workload of all the tasks that are released at time $t$, this value is equal to $\zeta = \max\{0, t - \sum_{i=0}^{n} \left\lfloor \frac{t - O_i}{T_i} \right\rfloor C_i\}$.

- To find an idle time after a time $t$, we have to add to the workload, the number of times when the processor was idle before. Doing so we get the real configuration of the system at time $t$.

- In conclusion, the first time when the processor will be idle after $t$, is the first time $t'$ after $t$ which respects the following condition $t' = \zeta + \sum_{i=0}^{n} \left\lfloor \frac{t' - O_i}{T_i} \right\rfloor C_i$.

In comparison to other methods, this method has the advantage that it doesn’t need any schedule, it is just based on a mathematical computation.

6.4 Finding the first periodic pattern of period

In fact we can reduce the study test to just look for the first idle time after $\min\{\left\lfloor \frac{O_{\text{max}} - O_i}{T_i} \right\rfloor T_i + O_i\} + P$ instead of $O_{\text{max}} + P$.

- The reason why we use $O_{\text{max}}$ in the computation of the bound is because between $O_{\text{max}}$ and $O_{\text{max}} + P$ we are sure that there is the cycle of Period. If it was before, we cannot be sure that the condition will be respected because of the non activation of the first job of some tasks.

- We can reduce $O_{\text{max}}$ by using instead a value $t' = \min\{\left\lfloor \frac{O_{\text{max}} - O_i}{T_i} \right\rfloor T_i + O_i\}$ which represents the earlier last activation of a task in the task set $\tau_n$, that means the first time when the cycle of Period of size $P$ begins. This value is shown in Figure 3.

- $t' \leq O_{\text{max}}$ and $t' = O_{\text{max}}$ appears in the case of a simultaneous start.

**Exemple.** Let’s consider three tasks $\tau_1, \tau_2, \tau_3$ where:

- $\tau_1$: $O_1 = 0, T_1 = D_1 = 9, C_1 = 2$
- $\tau_2$: $O_2 = 4, T_2 = D_2 = 9, C_2 = 2$
- $\tau_3$: $O_3 = 8, T_3 = D_3 = 9, C_3 = 2$

$$t' = \min\{\left\lfloor \frac{O_{\text{max}} - O_i}{T_i} \right\rfloor T_i + O_i\}$$

$$= \min\{8 - 0, \frac{8 - 4}{9}, \frac{8 - 8}{9}\} = 0$$

![Figure 3: Reducing $O_{\text{max}}$](image)

Hence, the periodic pattern of period of size $P$ begins at time 0, which is lower than $O_{\text{max}} = 8$. This example is illustrated in Figure 4.

![Figure 4: Example $O_{\text{max}}$](image)

7. SIMULATION

7.1 Simulation Setup

In this section, we present the setup of our simulation. In the following, we fix some constraints:

- The number of phases of one task will be limited to 6 to limit the simulation time.
- The parallel overhead (vector of job parallelism) is equal to $\gamma_i,j = \left\lfloor \frac{C_{i,j}}{T_i} \right\rfloor$.
- Two successive phases in a task are never of the same type, i.e. if the first one is a parallelizable phase, the second is sequential and conversely.

**Generation of a parallel task.**

The task generation uses a two-step procedure. We first generate a simple task with only one phase. The second step decomposes the task into phases.

1. Step 1: Generation of a sequential task The first step consists in the generation of the sequential task $\tau_i$:

   (a) $T_i$ is uniformly chosen in $[1, 100]$;
   (b) $u_i$ (truncated between 0.001 and 0.999) is uniformly chosen in $[\frac{1}{T_i}, 1]$;
(c) $C_i = T_i \times u_i$;
(d) $D_i$ is uniformly chosen in $[C_i, T_i]$;

2. Step 2: Decomposition of the sequential task into phases
The second step consists in the decomposition of the previous generated task into phases:
(a) $\ell_i$ is uniformly chosen from $[1, \min(C_i, T_i)]$;
(b) For each phase from 1 to $\ell_i$, $C'_i$ is uniformly chosen in $[1, (C_i - \sum_{j=1}^{i-1} C'_j) - (\ell_i - j)]$

**Generation and tests of task sets.**

Task sets are generated such that those obviously unfeasible ($U > m$) or trivially schedulable ($m \geq n$ and $\forall i \in [1, n], u_i \leq 1$) are not considered during simulations.

The procedure to create and test task sets is as follows:
1. Initially, we generate a set which contains $m + 1$ tasks and test it.
2. Gradually, we add new tasks to the set and repeat the tests until the whole utilization of the system exceeds $m$ (the capacity of the identical processors platform).

**7.2 Results**

We consider 4-processor, 8-processor and 16-processor identical platforms and generate 10,000 task sets for each platform. First, we have tested if there was a domination constraint between all the three local deadline assignment methods. We conclude that no domination property could be found. It means that each method can schedule some task sets that others cannot. The simulations results are shown on Figures 5 and 6. We have compared in those simulations the effectiveness of three methods based on the number of schedulable systems and on the execution time to decide on the schedulability of the tasks w.r.t. local deadline assignment methods. Simulations on 4, 8 and 16 processors give similar results.

The scheduling algorithm based on ‘Minimum Local Deadline Assignment’ (called LocalFairAssignation in Figures) has a lower scheduling success ratio (about 10%) than the other two methods for low processor utilization. However when the total utilization of the system is very high (close to 1 for each processor), this algorithm schedules more task sets (about 20%) than the other two methods. For all utilizations, the scheduling based on ‘Local Set Deadline To The worst case response time’ schedules less task sets, for all processor utilizations, than the scheduling based on ‘Local Fair Deadline’ method.

Regarding the execution time to decide on the schedulability of a task set, schedules based on ‘Minimum Local Deadline Assignment’ method and ‘Local Fair Deadline’ method take a similar execution time for low and medium processor utilizations. For high processors utilizations, ‘Minimum Local Deadline Assignment’ takes much longer time than the other two methods. This is explained by the fact that this method requires to schedule more systems at this level of utilization, so it requires more computations. The scheduling based on ‘Local Set Deadline To The worst case response time’ method leads to more time than the other two methods, except when the utilization is close to the number of processors (since the algorithm schedules less systems, it takes less time).

**7.3 Conclusion**

We clearly see that the efficiency of local deadline assignment methods is inversely proportional to the associated time required to decide on the schedulability of a task set. The most interesting method is EDFLocalAssignation, and the worst is the SequentialPhaseAssignation method in term of success ratio and number of parallelization. The more we parallelize, the higher schedulable task sets.

**8. CONCLUSION**

In this article, we use a semi-partitioned approach for parallel scheduling of Multi-Phase Multi-Thread tasks. A task is composed of subtasks, each subtask having parallel phases and offsets. We revisit the results of Choquet-Geniet et al. [2] on uniprocessor offset scheduling. In their paper, the authors have shown that the scheduling of periodic tasks with offsets has transient and a permanent phases. They also give an algorithm to compute the duration of the transient phase. In our paper, we show that the permanent phase begins at time $t_{idle}$, the first idle time after time $O_{max} + P$ ($P$ is the least common multiple of periods). This leads, as in [2], to a schedulability interval $[O_{min}, t_{idle}]$ instead of $[O_{min}, O_{max} + 2P]$ that provides a 44% gain in simulation speed.

We propose a method to find the first idle time after any time $t$ to compute $t_{idle}$. We then propose a new sufficient schedulability condition for the uniprocessor EDF scheduling with offsets based on an adaptation of the demand bound function applied to a time interval of size $P$.

Finally, we propose three local deadline assignment heuristics for periodic Multi-Phase Multi-Thread tasks on a set of $m$ identical processors with Earliest Deadline First (EDF) scheduling. A local deadline is assigned to each phase. We study the success ratio of the EDF scheduling associated to each local deadlines assignment heuristic and the time to decide if a task set is schedulable. We shown that deadline assignment matters a lot for parallel real-time task systems, which, we hope, could help to further develop this line of work in the future.

**9. REFERENCES**

Figure 5: Results on 4 processors

(a) Number of schedulable tasks systems
(b) Execution time to decide if a task set is schedulable

Figure 6: Results on 8 processors

(a) Number of schedulable tasks systems
(b) Execution time to decide if a task set is schedulable


