Multiprocessor preprocessing algorithms for uniprocessor on-line scheduling*

Joël Goossens  
Université Libre de Bruxelles  
Joel.Goossens@ulb.ac.be

Sanjoy Baruah  
University of North Carolina  
baruah@cs.unc.edu

Abstract

Chetto & Chetto [2] presented an algorithm for the on-line admission control and run-time scheduling of aperiodic real-time jobs in preemptive uniprocessor environments that are executing systems of periodic hard-real-time tasks. This algorithm requires a significant degree of preprocessing of the system of periodic tasks – in general, this preprocessing takes time exponential in the representation of the periodic task system. In this paper, we develop techniques for speeding up the preprocessing phase of the Chetto & Chetto algorithm, by adapting it for implementation in parallel environments. We validate the effectiveness of our parallelization both by theoretical results, and through simulation experiments.

1 Introduction

Real-time systems are characterized by stringent timing constraints; hence, the correctness of a computation depends not only on its logical or computational results, but also on the instant when the result is made available. The most important feature of a real-time system is its predictability, i.e., the ability to determine whether the system is capable of meeting all its timing requirements. Examples of such systems include the control of engines, traffic, nuclear power plants, time-critical packet communications, aircraft avionics and robotics. Typically, time-sensitive computations are modelled as jobs that need to be executed by the system, and the timing constraint is represented by a deadline denoting the time-instant by which the job should complete execution. We distinguish between two kinds of deadlines: If meeting a task deadline is absolutely critical for the system correctness, then the deadline is said to be hard; missing a hard deadline is considered a definite failure, and leads to catastrophic consequences. If it is desirable to meet a deadline but missing it can be tolerated, then the deadline is said to be soft.

In this paper, we consider the scheduling of periodic and aperiodic jobs. Periodic jobs are generated by periodic tasks: each periodic task \( \tau_i \) is characterized by an ordered pair of non-negative integers \((T_i, C_i)\) with \(0 < C_i \leq T_i\), i.e., by a period \( T_i \), and an execution requirement \( C_i \). Each such task generates an infinite sequence of jobs, with the \( k \)'th job generated by \( \tau_i \) arriving at time-instant \((k - 1) \cdot T_i\), and needing to execute for a total of \( C_i \) time-units by a (hard) deadline at time-instant \( k \cdot T_i \). Aperiodic jobs may arrive at any instant, and are characterized by an execution requirement and a deadline (which is usually not hard).

Scheduling analysis of real-time systems comprised entirely of periodic tasks has been extensively studied, primarily in the context of uniprocessor scheduling – the scheduling of real-time application systems where all the jobs with hard deadlines are constrained to execute on a single shared processor. For example, [11, 9, 10] have designed scheduling and feasibility-analysis algorithms for uniprocessor systems of periodic tasks in a preemptive environment – an environment in which a job executing on the processor may be interrupted at any instant and its execution resumed later, with no cost or penalty; in [6], feasibility-analysis of systems of periodic tasks in non-preemptive environments is considered. Scheduling of mixed systems composed of periodic tasks and aperiodic jobs has also been studied; see, e.g., [8, 7].

The Chetto & Chetto algorithm. Among the more influential bodies of research regarding the scheduling of such mixed systems in preemptive uniprocessor environments has been the work of Chetto & Chetto [2]. In [2], Chetto & Chetto considered the following scenario: Given a preemptive uniprocessor upon which it is known that a system of hard-real-time periodic tasks can be scheduled such that all jobs of all the tasks meet their deadlines, an aperiodic job, characterized by an arrival time, an execution requirement, and a deadline, is admitted if and only if it can be successfully completed by its deadline without compromising the feasibility of the periodic jobs and previously-admitted aperiodic jobs. Chetto & Chetto studied the on-line admission control problem for such mixed systems.
systems – how do we determine at run-time whether to admit a particular aperiodic job? To solve this problem, Chetto & Chetto proposed a preprocessing algorithm for determining the maximum quantity of idle time left available by the periodic tasks and previously-admitted aperiodic jobs between any two time-instants, and developed bookkeeping techniques for updating these quantities at run-time as new aperiodic jobs get admitted. On-line admission-control of aperiodic jobs is then easily implemented – when an aperiodic job arrives at time $a$ with an execution requirement $e$ and deadline $d$, it is admitted if and only if the idle time available in the interval $[a,d)$ is $\geq e$; if the job is admitted, then the available idle times are appropriately decremented. This approach has since been significantly extended to more general job, task, and system models, notably in the slot-shifting work of Fohler and colleagues (see, e.g., [3]).

This research. The preprocessing phase of the Chetto & Chetto algorithm requires the sequential consideration of a time-interval of length proportional to the least common multiple of the periods of the periodic tasks (see [2] for details), and is hence an inherently exponential-time operation. The emergence of cluster computing, in which “server farm” networks of computers based upon COTS processors and industry-standard software such as Virtual Interface Architecture, provide enormous computing power relatively inexpensively, suggests exploiting the parallelism capabilities of these computing clusters to tackle such useful problems that are difficult to solve efficiently within a traditional framework. We have been exploring the design, implementation, and analysis of efficient parallel algorithms for performing scheduling-theoretic analysis for uniprocessor real-time scheduling problems that take exponential time, or are provably intractable – NP hard – in the context of sequential computation. (Note that our goal is to use these distributed computing environments to solve uniprocessor scheduling problems: in this research, we are not addressing the issue of designing algorithms – distributed or sequential – for multiprocessor scheduling.) In [4], we have reported the results of our efforts regarding the feasibility-analysis of systems of asynchronous, periodic real-time tasks with distinct deadlines – a problem that is known to be co-NP-complete in the strong sense. In this paper we shall consider the application of our techniques to the scheduling of mixed systems comprised of periodic tasks and aperiodic jobs. In particular, we will consider the framework of Chetto & Chetto developed in [2] for the scheduling of mixed tasks under the earliest deadline first scheduling discipline (EDF) [11]. We shall propose a parallel implementation of this framework, and will show that this implementation is easily extended to be applicable for more general job and task models than just the one considered by Chetto & Chetto and a large family of scheduling algorithms (including EDF), with no increase in run-time computational complexity.

Our goal in this research is to design algorithms for use in relatively small computing clusters (with the number of processors $k$ a constant – such computing clusters are easily set up in small laboratories, using existing networked machines), and to obtain speedups that are as close to $k$ as possible in respect to the Amdahl’s Law [2]. Chetto & Chetto algorithm [2] implicitly simulates the scheduling of the system of periodic tasks using the EDF scheduling algorithm over a time interval of size exponential to the representation of the system of periodic tasks. The very notion of simulation is inherently sequential — in order to be able to simulate the behaviour of the system at instant $t$, it is necessary that we know the state of the system just prior to instant $t$. But in order to obtain this state, we must have simulated the behaviour of the system until instant $t$! We have however identified a sufficient set of conditions such that the simulation can be parallelized.

Organization of this paper. The remainder of this paper is organized as follows. In Section 2, we introduce some of the terminology and notation that will be used in later sections, and briefly describe popular task models and scheduling algorithms that we will consider for parallelization. In Section 3, we examine the sequential algorithm of Chetto & Chetto and we propose a reasonable uniprocessor implementation of their procedure. In section “Theoretical foundations”, we develop the theoretical foundations that form the basis of our parallelization techniques, by establishing a series of results concerning the behavior of schedules. We use this theory to design a parallel idle times determination algorithm in section “Parallelizing the idle slots determination”. In section “Synchronous periodic systems”, we apply the techniques of section “Parallelizing the idle slots determination” to a specific case – the idle times determination of synchronous periodic task systems when scheduled using the earliest deadline first scheduling algorithm – and further evaluate the parallelization of the idle times determination of synchronous periodic tasks, by validating by experimental evaluation in section “Experimental evaluation” the worst-case bounds that we establish in section “Synchronous periodic systems”.

2 Model and assumptions

Definition 1 A uniprocessor scheduling algorithm is defined to be expedient if the processor is never idled while there is an active job waiting to be executed. (Expedient scheduling algorithms are also called work-conserving al-
algorithms, or algorithms that use no inserted idle time in the literature.)

**Definition 2** A scheduling algorithm is said to be deterministic if the scheduling decision made by it depends upon only the current state of the system. Equivalently, the jobs active at a particular instant \( t \) (and the value of \( t \)) determine univocally the scheduling decision made by the algorithm at instant \( t \).

**Definition 3** A scheduling algorithm is said to be reasonable if it satisfies the following property: if it schedules a collection of hard-real-time jobs such that all the jobs complete by their respective deadlines, then it will schedule all subsets of this set of jobs such that all jobs complete by their respective deadlines.

All popular uniprocessor real-time scheduling algorithms that we are aware of are expedient and deterministic — the list of such algorithms includes earliest deadline first [11], rate monotonic [11] and deadline monotonic [10], and least laxity [12]. The preemptive versions of all these algorithms are reasonable as well; however, non-preemptive algorithms that are expedient and deterministic need not, in general, be reasonable.

Our approach to speeding up idle times determination on parallel machines is to essentially parallelize the simulation over a time interval; unfortunately, we have already seen that this process is not always parallelizable. So when can we parallelize the simulation? In order to answer this question, we need an additional definition.

**Definition 4** An idle instant occurs at instant \( t \) in a schedule if all jobs arriving strictly before \( t \) have completed execution before or at time \( t \) in the schedule.

Let \([t_0,t_k]\) denote the time interval for a particular simulation, and let \( t_1, t_2, \ldots, t_{k-1} \) denote distinct idle instants in the schedule, \( t_i < t_{i+1} \) for \( 0 \leq i < k \). If the scheduling algorithm is deterministic and expedient, we will prove that each of the \( k \) intervals \([t_0,t_1), [t_1,t_2), \ldots, [t_{k-1},t_k)\) may be simulated in parallel. Of course, this approach will provide no speedup if there are no idle instants in the time interval: hence the techniques developed in this paper are useful only in idle time determination problems where there is likely to be idle instants within the time interval. Note that this is certainly the case in any environment in which the Chetto & Chetto approach is likely to be used — in order to be able to accommodate any aperiodic jobs, it is necessary that there be idle instants (indeed, idle intervals) in any schedule for just the periodic tasks.

For any expedient scheduling algorithm it is of course possible to sequentially determine the various idle instants occurring in a time interval; however, doing so could require a computation time proportional to the length of the interval. Thus attempting to \emph{a priori} identify the idle instants prior to the simulation is not of much interest if our goal is to speed up the simulation by means of parallelism.

**Task models.** The techniques that we present in this paper may be used to parallelize the simulation based idle time determination of a wide variety of task models when they are scheduled using expedient, deterministic, scheduling algorithms. As an example application of our technique, we will consider the problem of idle times determination of a set of \emph{synchronous implicit-deadline periodic} real-time tasks (i.e., the task model considered by Chetto & Chetto in [2]). The set is composed of \( n \) periodic tasks \( \tau_1, \ldots, \tau_n \). Each periodic task \( \tau_i \) is characterized by an ordered pair of non-negative integers \((T_i, C_i)\) with \( 0 < C_i \leq T_i \), i.e., by a period \( T_i \), and an execution requirement \( C_i \). Each such task generates an infinite sequence of jobs, with the \( k \)-th job generated by \( \tau_i \) arriving at time-instant \( (k-1)T_i \), and needing to execute for a total of \( C_i \) time-units by a (hard) deadline at time-instant \( k \cdot T_i \).

A more general periodic task model that has also been widely studied concerns \emph{asynchronous periodic} real-time tasks. Each asynchronous periodic task is characterized by the 4-tuple of non-negative integers \((T_i, D_i, C_i, O_i)\) with \( 0 < C_i \leq D_i, C_i < T_i \), and \( O_i \geq 0 \), i.e., by a period \( T_i \), a deadline delay \( D_i \), an execution time \( C_i \), and an offset \( O_i \), giving the arrival instant of the first job of the task. The jobs generated by such a task arrive at time-instants \( O_i + (k-1)T_i \) \( (k = 1, 2, \ldots) \); each job has an execution requirement of \( C_i \) units, and the \( k \)-th job has its deadline at \( O_i + (k-1)T_i + D_i \). We point out that although we focus primarily in the rest of this paper upon the synchronous implicit-deadline systems considered by Chetto & Chetto [2] (i.e., \( D_i = T_i \) for all tasks and \( O_i = O_j \forall i,j \)), our techniques cover the general case of asynchronous periodic tasks with arbitrary deadlines (no constraints exist between the deadline of a task and its period).

All timing characteristics of the tasks in our model of computation are assumed to be non-negative integers.

**3 Implementing the sequential algorithm of Chetto & Chetto**

The Chetto & Chetto framework can be considered to be comprised of two components: (i) A \emph{preprocessing} algorithm, during which certain data structures are initialized; these data structures permit the efficient determination of the maximum quantity of idle time left available by the periodic tasks between any two time-instants. (ii) An \emph{on-line} algorithm, which responds to the arrival of an aperiodic job at time-instance \( d \) with execution-requirement \( e \) and deadline \( d \) by using these data structures to determine whether
the amount of idle time left available by the periodic tasks and previously-admitted aperiodic jobs during \([a_i, d_i]\) is \(\geq e\); if so, the aperiodic job is admitted and the data structures updated appropriately. The preprocessing algorithm, as defined by Chetto & Chetto [2], takes time exponential in the representation of the system of periodic tasks – it is this algorithm that we are seeking to parallelize.

In this section we shall examine the (uniprocessor) implementation of the preprocessing procedure as defined in [2]. We shall see that while this procedure is theoretically well defined, its implementation raises some interesting questions. But, let us first briefly review the Chetto & Chetto procedure (see [2] for further details).

Consider the system of \(n\) periodic tasks \(\tau_1, \ldots, \tau_n\), with \(\tau_i = (T_i, C_i)\), for \(1 \leq i \leq n\). Let \(P = \text{lcm}\{T_i \mid i = 1, \ldots, n\}\) denote the hyper-period of the system and \(U = \sum_{i=1}^{n} \frac{C_i}{T_i}\) the utilization factor of the system, i.e., the (asymptotic) fraction of the processor time spent in the execution of the task set, if feasible.

For synchronous periodic real-time task systems it has been shown [11, 5] (under some conditions) that the EDF schedules are periodic from 0 with a period of \(P\). For this reason idle time determination during the preprocessing phase may be restricted to the interval \([0, P]\).

The procedure in [2] constructs first the set \(E = \{r | r = k \cdot T_i, 0 \leq k < e_i, i = 1, \ldots, n\}\) (the job-arrival instants for all periodic jobs) then constructs a “sorted” version of \(E\) the vector \(E = (e_0, \ldots, e_p)\) with \(e_i < e_{i+1}\) and \(e_0 = 0\), where \(p = \#E\), the total number of distinct time-instants in \([0, P]\).

In [1] it was proved that, if an idle time-interval exists when the system of periodic tasks is scheduled using EDF, then this idle time interval immediately precedes an instant in \(E\). The procedure proposes then to compute the length of these idle times: \(\Delta_j\) denotes the length of the idle time interval which immediately precedes the instant \(e_j\) and can be computed by the following iterative process: \(\Delta_0 = 0, \Delta_i = \max\{0, e_i - \sum_{j=1}^{i-1} \frac{C_j}{T_j} - \sum_{k=1}^{i-1} \Delta_k\} \quad 0 < i \leq p\).

For given \(n\), the sequential complexity of this procedure for computing the \(\Delta_j\)'s is \(O(R)\) where \(R = \sum_{i=1}^{n} e_i T_i\), i.e., the total number of periodic job arrivals in \([0, P]\). In general (particularly when \(P\) is very large — note that \(P\) may be exponentially large in terms of the representation of the periodic task set) it is not possible to construct the vector \(E\) in time linear in \(P\); furthermore, it is not reasonable to assume that the quantities \(e_0, \ldots, e_p\) remain in main memory. Therefore, a direct implementation of the possible algorithm in [2] would require the use of a very large file to store the set \(E\), and use an external sorting algorithm (e.g., the polyphase merge algorithm) to compute the vector \(E\). The time complexity is \(\tilde{O}(R \cdot \log \frac{D}{m})\), where \(m\) denotes the size of the main memory used by the polyphase merge algorithm.

Rather than resort to external sorting in this manner, our preferred approach is to instead use an event-driven simulator directly generate and store in a file the sequence: \(e_0, \ldots, e_p\). The maximal time complexity is \(O(R)\), the number of events of the event-driven simulation. Notice that the iterative procedure of Chetto & Chetto (described above) for computing the \(\Delta_j\)'s does not require that all the \(e_i\)'s be in main memory simultaneously — the \(i\)th iteration of the computation depends only upon \(e_i\) and the sum \(\sum_{j=1}^{i-1} \Delta_j\). Consequently we propose that in a sequential implementation, the event-driven simulator write to a file the time sequence \(e_0, \ldots, e_p\), while maintaining the sum \(\sum_{j=1}^{i-1} \Delta_j\) in main memory and hence simultaneously computing and writing out the quantities \(\Delta_1, \ldots, \Delta_p\).

4 Parallel implementation

We have briefly described in the previous section our preferred sequential implementation of the procedure of Chetto & Chetto to compute the idle times. It may be noticed that the iterative process for computing the \(\Delta_j\)'s, cannot be directly parallelized since computing \(\Delta_j\) requires knowledge of the sum \(\sum_{j=1}^{i-1} \Delta_j\). We shall develop in this section a new algorithm for computing the idle times which is parallelizable and can be applied not only for EDF-scheduled synchronous implicit-deadline periodic task systems but for a very large class of scheduling algorithms (including EDF) and task models. We will see that the parallel algorithm will involve very little inter-processor communication, thus rendering it particularly suitable for implementation on clusters of computers.

Theoretical foundations In this section we develop the theoretical foundations upon which our idle times determination will be developed. Throughout this section we will be assuming that our scheduling algorithm is expedient and deterministic (although not necessarily reasonable, unless explicitly stated to be so).

We will define a schedule for a given set of jobs \(J\) to be a function from the non-negative integers – denoting time slots – to the set \(J \cup \{\perp\}\): If the processor is executing job \(x\) during \([t, t+1]\), then \(\sigma(t) = x\); while if the processor is left unassigned during \([t, t+1]\), then \(\sigma(t) = \perp\). We will assume that a scheduling algorithm does not stop even when a deadline is missed, but rather continues executing according to some deterministic and expedient scheduling rule. That is, we define the function \(\sigma(t)\) for all \(t \geq 0\) even if deadlines are missed – if a job misses its deadline, we assume that the execution of the system continues and the job...
(which has missed its deadline) remains active until its completion. In case a deadline is missed, we leave unspecified how the scheduling algorithm makes scheduling decisions, other than that it be (deterministic and) expedient – i.e., the CPU not be left idle while there are active jobs awaiting execution (before or after their deadlines).

Below, we will characterize the occurrence of idle instants in schedules that are generated by any expedient algorithm. But first, a definition.

**Definition 5** The demand of a set of jobs at time $t$ is defined to be the sum of the computation times of all the jobs that arrive strictly before time $t$.

**Lemma 6** Let $J$ be an arbitrary set of jobs and $\sigma$ its schedule using a scheduling algorithm $A$. Suppose that at some instant $t$ there is an idle instant in $\sigma$. Let $J'$ be a subset of the jobs in $J$ ($J' \subseteq J$), and $\sigma'$ be the schedule obtained by scheduling $J'$ using any expedient scheduling algorithm $B$. The instant $t$ is an idle instant in the schedule $\sigma'$.

**Proof.** First, it may be noticed that at each instant the demand in $\sigma$ is greater than (or equal to) the one in $\sigma'$, since $\sigma'$ is obtained by dropping some jobs in the original system $J$. At time $t$ in $\sigma$ all the demand has been served ($t$ is an idle instant in $\sigma$), since the scheduling algorithm $B$ is expedient it follows that at time $t$ the demand in $\sigma'$ must be also served, the property follows.

Recall that our parallelization strategy, as outlined in Section 2, was to simulate the behavior of the scheduling algorithm on the task system in parallel between every set of idle instants. Unfortunately, we cannot efficiently determine the idle instants a priori, and hence cannot do this directly. The following theorem proves, however, that we can nevertheless begin several parallel simulations without knowing the idle instants beforehand, although the validity of any individual parallel simulation (other than the one that simulates beginning at the start of the time interval) may not be known until another of the parallel simulations has validated it. Informally, Theorem 7 below essentially states the following: Partition the time interval into contiguous intervals. Begin simulating the behavior of the system on each of these intervals in parallel, and let $\sigma_j$ denote the schedule obtained by the $j$'th simulation. This simulation is to continue beyond the end of the interval and into the next contiguous interval(s), until an idle point $I_j$ is reached that lies after the end of the interval. Assuming that the $j$'th simulation has been validated to be correct prior to instant $I_j$, the schedule $\sigma_{j+1}$ agrees with the actual schedule – the one that would be generated in a sequential simulation starting from the beginning of the interval – after time-instant $I_j$.

**Theorem 7** Let $J$ be an arbitrary set of jobs and $\sigma$ its schedule using an expedient and deterministic scheduling algorithm. Let $t_0$ be the instant at which the first job in $J$ occurs (i.e., arrives). Let $t_1, t_2, \ldots$ be such that $t_0 \leq t_1 \leq t_2 \ldots$. Let $\sigma_j (0 \leq j)$ be the schedule obtained by the same scheduling algorithm by only considering in $J$ the jobs that occur at or after time $t_j$. Let $I_j (0 \leq j)$ be the first idle instant in the schedule $\sigma_j$ after time $t_{j+1}$. Let $\lambda_j$ be defined as follows: $\lambda_0 = t_o, \lambda_j = \max_{0 \leq k < j} \{I_k\},$ for $j > 0$. We have that for all $j \geq 0, t \geq \lambda_j \Rightarrow (\sigma_j(t) = \sigma(t))$.

**Proof.** By induction on $j$. The property is true in the base case, $j = 0$, since $\sigma_0(t) = \sigma(t)$ for all $t \geq \lambda_0 = t_0$.

Suppose the property is true until $r$: for all $j, 0 \leq j < r$, $t \geq \lambda_j \Rightarrow (\sigma_j(t) = \sigma(t))$.

Consider now the schedule $\sigma_r$, and let $t \geq \lambda_r$. That is, $t \geq \max_{0 \leq \ell < r} \{I_{\ell}\}$, from which it follows that $t \geq \max_{0 \leq \ell < r-1} \{I_{\ell}\}$, or equivalently $t \geq \lambda_{r-1}$.

By the induction hypothesis $t \geq \lambda_{r-1} \Rightarrow (\sigma_{r-1}(t) = \sigma(t))$. Hence $\lambda_r$, which by definition is at least as large as an idle instant $\geq t_r$ in $\sigma_{r-1}$, is at least as large as an idle instant $\geq t_r$ in $\sigma$. By Lemma 6, this idle instant in $\sigma$ is also an idle instant in $\sigma_r$. Since the scheduling algorithm is deterministic and expedient, it follows that all scheduling decisions made by the scheduling algorithm after this idle instant in $\sigma_r$ mimic the scheduling decisions made by the scheduling algorithm in $\sigma$; from which it follows that $\sigma_r(t) = \sigma(t)$.

Concerning the sequence of values $\lambda_0, \lambda_1, \ldots$, defined in Theorem 7 above we note that, with $\lambda_0$ defined to be equal to $t_0$, $\lambda_j$ for $j > 0$ may be inductively defined by $\lambda_j = \max \{\lambda_{j-1}, I_{j-1}\}$.

**Parallelizing the idle slots determination** Given a uniprocessor real-time system and a deterministic expedient scheduling algorithm for scheduling this real-time system, a simulation-based idle slots determination algorithm computes, by simulating the behavior of the scheduling algorithm on the real-time system, where the idle slots appear. In this section we will apply the theory developed in the previous section (“Theoretical foundations”) to parallelizing, for execution on the $k$ processors $P_{\alpha}, \ldots, P_{k-1}$, such simulation-based idle time determination.

Let $(t_{\alpha}, t_k)$ denote the interval within we would like to determine the idle times, we assume that $t_\alpha$ is an idle instant. We will choose $t_1, \ldots, t_{k-1}$ such that $t_\alpha < t_1 < \cdots < t_k$.

The $j$'th processor $P_j (0 \leq j < k)$ will simulate the behavior of the deterministic expedient scheduling algorithm on the jobs of the set of tasks that occur (arrive) at or after instant $t_j$ — we will refer to this simulation performed by processor $P_j$ as $\gamma_j$. We will see that it follows from Theorem 7 that this simulation $\gamma_j$ may be terminated after the simulation of the first idle instant after $t_{j+1} - \text{instant } I_j$ of Theorem 7 — has been reached.

It follows directly from Theorem 7 that a idle slot in $\gamma_j$ is an idle slot in the original schedule $\sigma$ if this one occurs after
or at time $\lambda_j$. Of course for all $j > 0$ processor $P_j$ has no a priori knowledge of the value of $\lambda_j = \max(\{\lambda_{j-1}, I_{j-1}\}$; this value can only be computed after $\lambda_{j-1}$ and $I_{j-1}$ have been communicated to $P_j$ by processor $P_{j-1}$. It may be noticed that in general it is not reasonable to store the various idle times in main memory, since the number of idle time intervals may be proportional to $P$. For this reason we propose that each processor (except $P_0$) operate in two passes. In the first pass $P_j$ simulates the behavior of the deterministic and expedient scheduling algorithm, terminating its simulation at (simulated) time $I_j$ and storing in a file (say $F_j$) the various idle times. In the second pass, $P_j$ receives $\lambda_j$ from $P_{j-1}$, communicates $\lambda_{j+1}$ to processor $P_{j+1}$ and only considers in the file $F_j$ those idle times included in the interval $[\lambda_j, \lambda_{j+1})$. Processor $P_0$ knows the value of $\lambda_0$ beforehand (since $\lambda_0 = t_0$ by definition).

To summarize: (i) Initially, the values of $t_j$ and $t_{j+1}$ are communicated to processor $P_j$; (ii) In addition, the parameters of all jobs of the real-time system that are generated at or after instant $t_j$ must be communicated to processor $P_j$; (iii) $P_j$ simulates the behavior of the (deterministic and expedient) scheduling algorithm on its set of jobs, and writes to the file $F_j$ the idle times identified, until an idle instant $I_j$ at or after $t_{j+1}$ is reached during the simulation; (iv) Processor $P_j$ knows the value of $\lambda_j$ beforehand (since $\lambda_j = t_0$ by definition); for all $j > 0$, $P_j$ will receive the value of $\lambda_j$ from $P_{j-1}$; (v) Once $P_j$ knows $\lambda_j$ and $I_j$, it computes $\lambda_{j+1} = \max(\{\lambda_j, I_j\}$, and communicates this value to $P_{j+1}$. Henceforth, $P_j$ will only consider in the file $F_j$ those idle times included in the interval $[\lambda_j, \lambda_{j+1})$, and purges the remaining entries — those covering times $< \lambda_j$ or $> \lambda_{j+1}$ — from $F_j$.

**Synchronous periodic systems** In order to illustrate the techniques we have developed in this paper we will now parallelize, for execution on a cluster of $k$ computers ($k > 1$), the idle times determination of $n$ synchronous periodic real-time tasks (Section 2) $T_1, \ldots, T_n$ (i.e., the task model considered by Chetto & Chetto). Each periodic task $T_i$ is characterized by the ordered pair of non-negative integers $(T_i, C_i)$ with $0 < C_i \leq T_i$, i.e., by a period $T_i$, an execution time $C_i$. Recall that in this case all jobs start their execution at the same time and without loss of generality we shall assume that this time is equal to zero and consequently the origin is an idle instant. Below, we first show that when the degree of parallelism is not the constraining factor — i.e., when the cluster contains an unbounded number of computers — the length of the longest busy period is an upper bound on the run-time of our cluster-based simulation algorithm. Then, we consider the situation when the number of computers in the cluster is limited.

**Lemma 8** The simulation $\gamma_j$ ($j < k - 1$) ends at or before time $t_{j+1} + L^m$, where $L^m$ is the length of the largest busy period, provided the scheduling algorithm used is expedient and deterministic.

**Proof.** The simulation $\gamma_j$ terminates upon simulating the first idle instant after time $t_{j+1}$. Since the length of the largest busy interval is $L^m$, there is guaranteed to be such an idle instant in the interval $[t_{j+1}, t_{j+1} + L^m)$; the simulation $\gamma_j$ will therefore encounter its desired idle instant and terminate prior to $t_{j+1} + L^m$. The property follows from Lemma 6.

The last simulation $\gamma_{k-1}$ ends its execution at most at time $t_k$.

We know from [11, 5] that $L^m$ is bounded by the length of the first busy period in the synchronous case and it is also the smallest positive solution to the equation: 

$$L^m = \sum_{i=1}^{n} \left[ T_i, C_i \right] \times C_i,$$

which can be solved by a simple iterative process. (Observe that while this result was proven in the literature for the deadline driven scheduler EDF, it is not difficult to see that it holds for all expedient algorithms.)

Note that $L^m$ is bounded by $\sum_{i=1}^{n} \frac{C_i}{T_i}$ (since $\frac{T_i}{C_i} \leq 1 + \frac{L^m}{t_f}$).

**Theorem 9** The maximal parallel time complexity of our algorithm for determining idle times in synchronous systems scheduled using an expedient and deterministic scheduling algorithm is $O(L^m + \max\{t_{j+1} - t_j \mid j = 0, \ldots, k - 1\})$.

**Proof.** Immediately follows from Lemma 8.
cases: (i) the utilization factor is \( \approx 1 \), in this case \( L^m \) is \( \approx P \), and the improvement is negligible; (ii) the utilization is very small (i.e., near zero), the improvement is exponential but the numbers of computers needed in the cluster is also exponential. But if we have available \( k \) computers in the cluster, we could choose for instance \( d = \left\lceil \frac{L^m}{k} \right\rceil \), in this case the maximal time complexity of our idle times determination is \( O\left(\frac{P}{k} + L^m\right)\).

**Experimental evaluation** The experiments described in this section, which were performed to estimate the efficacy of our proposed parallelization technique for obtaining significant speedups while performing idle times determination of real-time systems, were actually implemented on a uniprocessor machine – a Ultra SPARC 10 machine, UltraII 330 MHz processor, with 640 Mbytes RAM – which emulated the behavior of a computing cluster. (Since our parallel idle times determination algorithm involves very little inter-processor communications, our simulation ignores the time taken for such communication.)

We are cognizant that it is in general very difficult to draw accurate conclusions regarding the benefits of a proposed technique from simulations, since these benefits often depend in a non obvious way upon the many parameters of the real-time system, in particular on the (distribution of the) system characteristics (the number of jobs, the load of the system, etc). It is of course not possible to consider all distributions of real-time systems in our simulations; moreover, it is difficult to determine which distributions are (perhaps) realistic. For our simulation experiments, we have therefore made use of the pseudo-random task set generator developed by Ripoll et al. [13], which they have very generously made available to us – since workloads generated by the Ripoll et al. generator have been previously used for performing feasibility-analysis experiments and these experiments have been revealed to the larger research community for several years now, we believe that using this task generator provides a context for our simulation results, and allows them to be compared with other results performed by other researchers.

As stated above, we have restricted our experiments to systems composed of synchronous periodic task sets. We have simulated the scheduling of such systems using the expedient, deterministic, and reasonable EDF scheduling algorithm. Our experimental methodology is as follows. We limit our study to randomly chosen task sets, our goal being to have an indication of the actual performance of our idle times determination. We use the pseudo-random task set generator proposed by Ripoll et al. [13] with the same parameters. With this generator we generated a large number of periodic task sets, with the same parameters chosen by Ripoll et al. – the computation times are uniformly chosen from the interval \([1, 20]\), and the periods from the interval \([3, 670]\); the utilization factor is uniformly drawn from the interval \([0.3, 0.95]\). Since parallelization for obtaining speedups is an issue only if the sequential approach would take inordinately long, we only consider those task sets from among the generated task-sets that have a hyper-period in the range \((10^3, 10^9)\) – i.e., between 100 million and one billion. We performed our experiments on a uniprocessor system — a computing cluster was simulated by emulating in turn the behavior of each of the computers that comprise the cluster. We measure the time needed by our parallel idle times determination as follows: each of the computers comprising the cluster would have executed to completion, and the time taken would be equal to the the maximum of the times taken to simulate the individual computers.

Let \( t_{\text{Chetto}} \) denote the time needed by the Chetto & Chetto framework – Figure 1 shows the distribution of \( t_{\text{Chetto}} \) as measured in our experiments. Figure 2 shows the distribution of the time needed by our idle times determination using a cluster of 10 computers and for the same task sets – we refer to this time as \( t_{\text{multis}} \). Figure 3 shows the distribution of the ratio \( \frac{t_{\text{multis}}}{t_{\text{Chetto}}} \) – the mean ratio is seen to \( \approx 16 \), with standard deviation \( \approx 5 \); from which we conclude that on average our parallelization of the feasibility test results in a speedup of nearly 16. Observe that this is not a contradiction with
the Amdahl’s Law since $t_{Chetto}$ is not the time of the fastest sequential algorithm to resolve the problem and our parallel algorithm is not a strict parallelization of the Chetto & Chetto framework but a parallelization of an event-driven simulation.

We also noticed that on average the speedup grows (i.e., is better) when the utilization is larger.

5 Conclusions

In this paper, we studied the problem of determining on line whether to admit an aperiodic real-time job in a preemptive uniprocessor environment that is executing a system of periodic hard-real-time tasks, such that the feasibility of none of the periodic jobs or previously-admitted aperiodic jobs is compromised. One approach towards solving this problem was pioneered by Chetto & Chetto [2]; this approach has subsequently been extended by Fohler and colleagues [3]).

The Chetto & Chetto approach (and hence by extension, the work of Fohler et al.) suffers from the drawback that it requires exponential-time preprocessing of the system of periodic tasks, in order to compute certain data structures that are necessary for performing on-line admission control. In this paper, we have described our research efforts at ameliorating this computational complexity by developing a simulation-based variant of the Chetto & Chetto preprocessing algorithm (Section 3), and then applying parallelization techniques to parallelize the simulation. We have developed the theoretical foundations ([4]) that permit such parallelization of a problem significantly more general than the one considered in [2], and have used this theory to parallelize the Chetto & Chetto algorithm. We have validated the viability of our techniques by implementing our parallel version of the Chetto & Chetto algorithm, and have performed a series of simulation experiments to measure the kinds of speedups that are obtained by such parallelizations. As future work, we plan to (i) implement our algorithm on actual computing clusters, and (ii) explore the possibility of extending our implementation to clusters comprised of very large numbers of physically distributed computers, which are connected to each other over the Internet.

References