Experimental Analysis of the Tardiness of Parallel Tasks in Soft Real-time Systems

Manar Qamhieh, Serge Midonnet

Experimental Analysis of the Tardiness of Parallel Tasks in Soft Real-time Systems

Manar Qamhieh & Serge Midonnet
{manar.qamhieh,serge.midonnet}@univ-paris-est.fr

Université Paris-Est, France

Abstract. A parallel application is defined as the application that can be executed on multiple processors simultaneously. In software, parallelism is a useful programming technique to take advantage of the hardware advancement in processors manufacturing nowadays. In real-time systems, where tasks have to respect certain timing constraints during execution, a single task has a shorter response time when executed in parallel than the sequential execution. However, the same cannot be trivially applied to a set of parallel tasks (taskset) sharing the same processing platform, and there is a negative intuition regarding parallelism in real-time systems. In this work, we are interested in analyzing this statement and providing an experimental analysis regarding the effect of parallelism soft on real-time systems. By performing an extensive simulation of the scheduling process of parallel taskset on multiprocessor systems using a known scheduling algorithm called the global Earliest-Deadline First (gEDF), we aim at providing an indication about the effects (positive or negative) of parallelism in real-time scheduling.

Keywords: parallelism, stretching techniques, real-time systems, soft real-time systems, scheduling simulation, global earliest deadline first.

1 Introduction

Uniprocessor platforms have been widely used in computer systems and applications for a long time. However, making processors smaller and faster has become more challenging for manufacturers recently due to the physical constraints such as heating and power problems. As a result, manufacturers are moving toward building multicore and multiprocessor systems so as to overcome these physical constrains. In the last few years, we have witnessed a dramatic increase in the number of cores in computational systems, such as the 72-core processor of the TILE-Gx family from Tilera, and the 192-core processor released by ClearSpeed in 2008.

Unfortunately, there is a gap between the advancement in software and hardware, and most of the currently used applications are still designed to target uniprocessor systems as execution platforms [1]. In order to get full advantage of multicore and multiprocessor systems, parallel programming has been employed

so as to perform computations and calculations simultaneously on multiple processors. Lately, it has gained a higher importance although it has been used for many years.

The concept of parallel programming is to write a code that can be executed simultaneously on different processors. Usually, these programs are harder to be written than the sequential ones, and they consist of dependent parts. However, global scheduling, in which the scheduler can execute any job on any available processor at any time instant, is more suitable for parallel programs than partitioned scheduling (in which jobs are assigned first to individual processors and are forced to execute without processor migration).

From practical implementation's point of view, there exist certain libraries, APIs and models created specially for parallel programming like POSIX threads[2] and OpenMP [3]. Except these are not designed normally for real-time systems. In embedded systems, software usually is subjected to certain timing constraints, such as operational deadlines and the frequency of job arrivals. These constraints affect the correctness of its results along with the correctness of the calculations, and these systems are referred to as real-time systems.

Based on the criticality of the timing constraints, real-time systems are classified as either hard or soft. In hard real-time systems, the consequences of deadline misses can cause catastrophic effects, while soft real-time systems can tolerate delays in the execution of tasks, and a deadline miss only degrades the quality of service provided by the application. Avionic and transportation systems are examples of hard real-time systems, while communication and media systems (such as in video surveillance) are considered as soft real-time systems.

In real-time systems, many researches focused on the sequential task model in the case of multiprocessor systems [4]. In comparison, Few studies are conducted on the different models of parallel tasks, such as the multi-threaded segment and the Directed Acyclic Graph (DAG) model [5, 6]. Mainly, a parallel task model is divided into three categories:

- rigid if the number of processors is assigned externally to the scheduler and can't be changed during execution,
- moldable if the number of processors is assigned by the scheduler and can't be changed during execution (this is the model we consider in this paper),
- malleable if the number of processors can be changed by the scheduler during execution.

In real-time systems, it has been believed that parallelism has negative effect on the schedulability of tasksets, as it has been stated in [7]. In that paper, the authors proposed to stretch the parallel tasks of the Fork-join model[1] as a way to avoid parallelism, and their results encourage the execution of parallel tasks as sequentially as possible. The reason why this was possible is that *fork-join tasks*

[1] A fork-join model is parallel task model in which the incoming jobs are split on arrival for service by numerous processors and joined before departure. It is the base of the OpenMP parallel programming API.

have schedulable utilization bounds slightly greater than and arbitrarily close to uniprocessors[2].

In this work, we aim at providing a basic intuition regarding the validity of this assumption, and we will study the effect of parallelism on the scheduling of tasksets on multiprocessor soft real-time systems. We use experimental analyses through extensive simulations as an indication tool towards our purpose. By choosing soft real-time systems, we can measure the performance of parallel tasks by calculating the tardiness of their jobs (how many time units a job needs after its deadline to complete its execution) in such systems when executed in parallel and sequentially, without worrying about the catastrophic effects of not respecting the timing constraints. To the best of our knowledge, we are not aware of similar researches or studies done regarding this assumption.

The structure of this paper is denoted as follows: Section 2 describes our parallel task model which is used in this paper, and it also includes two stretching transformations we propose in order to execute parallel tasks as sequentially as possible. These transformations are necessary as comparison references to the parallel execution. Then Section 3 contains a description about soft real-time systems and some details and discussion about an upper bound of tardiness found in literature. The contributions of this paper are included in Section 4, which provides a description about the simulation process used in this work, and the analysis of the conducted results. Finally, we conclude this paper by Section 5 which includes future work as well.

2 System model and its transformations

2.1 Parallel Task Model

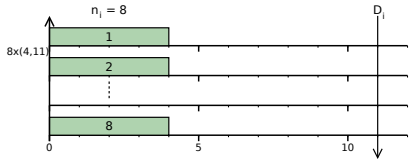
In this work, we consider a taskset τ that consists of n parallel sporadic implicit-deadline real-time tasks on a platform of m identical processors. Each task τ_i, where $1 \leq i \leq n$, is a parallel task that consists of a number of identical sequential threads, and it is characterized by (n_i, C_i, T_i), where n_i is the count of the threads belong to τ_i that can execute in parallel, C_i is the worst-case execution time (WCET) of τ_i which equals to the total WCET of its threads, and T_i is the minimum inter-arrival time (or simply the period) between successive jobs of the task, which defines the sporadic task. As in an implicit-deadline task, the deadline D_i of task τ_i, which is defined as the time interval in which τ_i has to complete its execution, is equal to the period of the task.

Let $\tau_{i,j}$ denote the j^{th} thread of τ_i, where $1 \leq j \leq n_i$. All the threads of the same task share the same period and deadline of their task. Let $C_{i,j}$ be the WCET of thread $\tau_{i,j}$, where $C_i = \sum_{j=1}^{n_i} C_{i,j}$. In this paper, we consider identical threads, however, the model can be generalized more by allowing non-identical threads.

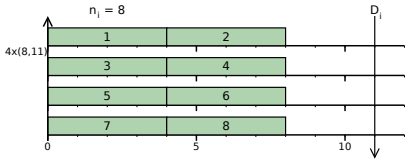
[2] The remark regarding the utilization of Fork-join tasks is quoted from [7]

the utilization of thread $\tau_{i,j}$ is defined as $U_{i,j} = \frac{C_{i,j}}{T_i}$, and the utilization of the parallel task τ_i is defined as $U_i = \frac{C_i}{T_i} = \sum_{j=1}^{n_i} U_{i,j}$.

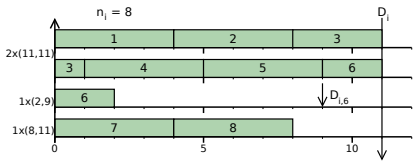
An example of our considered parallel task is shown in Figure 1(a), in which task τ_i consists of 8 threads and has a deadline and a period equal to 11.



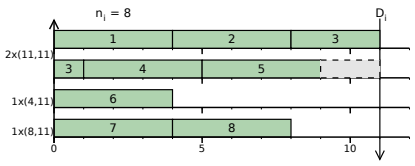
(a) All the threads of task τ_i execute in parallel.



(b) The threads of task τ_i execute as sequentially as possible without any migrations or preemptions due to transformation.



(c) All the threads of task τ_i execute as sequentially as possible. At most one constrained-deadline thread results.



(d) The transformation in Subfigure 1(c) is modified so as to get rid of the constrained-deadline thread.

Fig. 1. An example of a parallel task τ_i that consists of 8 threads and it has a deadline and period equal to 11. The rectangles represent the threads of the task, and the numbers within them indicate their indexes.

In real-time systems, a scheduling algorithm is defined as the algorithm that assigns priorities to the active jobs in the system, and it chooses which jobs can execute on the available processors at time t. If active jobs are authorized to migrate between processors, which means that a job can start its execution on one processor and then continue on another, then the scheduling algorithm is called global. In multiprocessor systems, few optimal algorithms[3] exist for global scheduling of tasksets. These algorithms suffer usually from high overhead costs consist of large number of jobs migrations and preemptions. However, there exist non-optimal algorithms that have good performance with lower costs such as the global earliest deadline first (gEDF), which we will use in this paper. The gEDF algorithm assigns the highest priority to the job with the earliest absolute

[3] An optimal algorithm is the one that can schedule successfully any feasible taskset. If the optimal algorithm fails in scheduling a taskset, then no other algorithm can schedule it.

deadline. It belongs to the fixed job priority family, in which the priority of the job is fixed during its execution but jobs of the same task have different priorities. Also we consider a preemptive scheduling, in which a higher priority job can interrupt the execution of a lower priority job, and the interrupted job can start its execution on a different processor.

Based on the categories of the parallel tasks in real-time systems, our considered parallel task model rigid w.r.t. the number of threads. A given task τ_i consists of n_i parallel threads is defined by the model. However, the execution behavior of the task depends on the number of available processors and the scheduling algorithm, and it is not obligatory to execute all the parallel threads together. So, these threads can execute either in parallel or sequentially based on the decisions of the scheduler. When a job is activated at time t, all the n_i parallel threads are activated as well. But if there exist less than n_i available processors at time t, then the scheduler executes partial set of the threads in parallel while the rest are executed later.

In this paper, we considered a simplified task model of parallelism, in order to better show the effect of parallelism on the scheduling of soft real-time systems. Our task model can be considered as a Fork-join task model, in which each task consists of one parallel segment, and the costs of fork and join events of the threads are included in the execution time of each thread. In the future, we aim to extend the work to include more realistic parallel task models such as the the multi-threaded task model and the Directed Acyclic Graphs (DAGs).

2.2 Stretching Parallel Tasks

In order to provide an analysis of the effect of parallelism on the scheduling of soft real-time systems, we will study the scheduling of a parallel taskset τ when executing on m identical processors using gEDF scheduling algorithm while considering some execution scenarios, that vary from parallel execution to sequential execution. As we described earlier in Section 2.1, the threads of a parallel task can execute either in parallel or sequentially based on the availability of processors and on the decisions of the chosen scheduling algorithm.

Hence, each parallel task τ_i in taskset τ can execute based on the following execution scenarios:

- the Parallel Scenario: all the threads τ_i execute in parallel, and they are activated at the same activation time of their parallel task τ_i (please refer to Figure 1(a)),
- the Fully-Stretched Scenario: all the threads of τ_i execute as sequentially as possible, and τ_i is transformed into a set of fully stretched threads and a thread is broken into at most two pieces which execute in parallel, while the stretched threads can be assigned dedicated processors (please refer to Figure 1(c)). A straight-froward transformation is provided in the following section in order to fully stretch the parallel threads.
- the Partially-Stretched Scenario: all the threads of τ_i execute as sequentially as possible, without causing interruptions and migrations due to transfor-

mation (please refer to Figure 1(b)). This transformation will be explained in more details in the following section.

The example of the proposed execution scenarios in Figure 1 might seem unclear now. We invite the reader to consult the following detailed sections regarding the execution scenarios while referring to the example in Figure 1.

The Parallel Scenario:
The Parallel scenario represents the default execution behavior of our parallel task model. According to this scenario, the threads of each parallel task are activated by the activation event of the original task, and they have its deadline and minimum arrival time between the jobs. Hence, all the threads have the same priority according to gEDF, and they have the same utilization. So, the scheduling of the parallel tasks on m identical processors can be treated as the scheduling problem of a taskset of $(n_i, \forall \tau_i \in \tau)$ sequential sporadic implicit-deadline threads. An example of the parallel scenario is shown in Inset 1(a), in which each thread of task τ_i has a worst-case execution time of 4 and a deadline equal to 11.

However, the maximum tardiness of a parallel task is determined by the maximum tardiness of all of its threads among all possible scenarios of jobs' activation.

Fully-Stretched Scenario:
The purpose of the Fully-Stretched transformation is to avoid the parallelism within the tasks when possible, by executing them as sequentially as possible. So, instead of activating all the parallel threads of a certain task at the same time (as in the Parallel scenario), this transformation determines which threads are activated in parallel and which are delayed to be executed sequentially. The objective is to transform the majority of the parallel threads of each task into fully-stretched threads which have a WCET equals to the period (utilization equals to 1). As a result, we can dedicate an entire processor for each transformed thread, which will guarantee their scheduling by the use of partitioned scheduling (tasks are assigned to a processor have to execute on this processor without authorizing migrations). Hence, the rest of the threads (not fully stretched) are the ones to be scheduled using gEDF, which will reduce their tardiness.

The Fully-Stretched Transformation is straight forward due to the simplicity of the considered parallel model. Let us consider a parallel task τ_i which consists of n_i threads and each thread $\tau_{i,j}$ has a WCET of $C_{i,j}$. The Fully-Stretched transformation will generate the following sets of modified threads $\tau'_{i,j}$:

- A set of fully-stretched threads $\tau'_{stretch}$: which consists of $\lfloor U_i \rfloor$ threads each has a total WCET equals to the original period, and utilization $U'_{i,j} = 1$, where $\tau'_{i,j} \in \tau'_{stretched}$. If the capacity of the processor cannot contain entire threads (i.e. $\frac{T_i}{C_{i,j}}$ is not an integer), then a thread will be forced to execute on 2 processors in order to fill the first one. As a result, the transformation will cause at most $\lfloor U_i \rfloor$ threads to migrate between processors.

– When the utilization of the parallel task is not integer, then there exist a set of threads that cannot fill an entire processor. Let the total remaining execution time be denoted as $C_{rem} = (U_i - \lfloor U_i \rfloor) * T_i$. The remaining threads are divided into the following two types:
  - At most, one implicit-deadline thread τ'_{imp} from each transformed parallel task is generated. This thread is created by merging the remaining threads that did not fit into the stretched tasks without the thread that is used to partially fill the last processor. The WCET of τ'_{imp} is calculated as $C_{imp} = (\lfloor \frac{C_{rem}}{C_{i,j}} \rfloor * C_{i,j})$, and it has a deadline and period equal to the original parallel task τ_i.
  - At most one constrained deadline[4] thread τ'_{cd} is generated, which has a WCET calculated as $C_{cd} = C_{rem} - C_{imp}$. Its period is the same as the original task τ_i, and it has a deadline calculated as $D_{cd} = (D_i - (C_{i,j} - C_{cd}))$. This thread contains the remaining execution time of the thread that had to fill the last stretched processor. The conversion from an implicit-deadline thread into a constrained deadline one is necessary to prevent the sequential thread from executing in parallel since its execution is divided between two processors.

Back to the example in Figure 1, Inset 1(c) shows an example of the Fully-Stretched transformation when applied on task τ_i shown in Inset 1(a). As shown in the figure, the original task consists of 8 threads each has a WCET equals to 4 and a deadline equals to 11. After transformation, the fully-stretched tasks $\tau'_{stretch}$ contains two threads. The first consists of threads $\tau_{i,1}$, $\tau'_{i,2}$ and the ending part of $\tau_{i,3}$. While the second task consists of the beginning part of the $\tau_{i,3}$ (complementary to the part in the previous task), threads $\tau_{i,4}$, $\tau_{i,5}$ and the ending part of thread $\tau_{i,6}$. The remaining execution time of thread $\tau_{i,6}$ forms the constrained deadline independent thread τ'_{cd}, with a deadline $D_{i,6} = 9$ as shown in the figure. Threads $\tau_{i,7}$ and $\tau_{i,8}$ are merged together, in order to form a single implicit-deadline task with a WCET equals to 8 and a deadline equals to 11.

The advantage of the Fully-Stretched Transformation is that, at most, two threads (τ'_{imp} and τ'_{cd}) are scheduled using gEDF, and they are the ones that may cause a tardiness during the scheduling process. While the rest of the generated threads ($\{\tau'_{stretch}\}$) are scheduled using partitioned scheduling algorithms, and they are guaranteed to respect their deadline each on a single processor independently.

Partially-Stretched Scenario:
A modification to the Fully-Stretched transformation can be proposed so as to avoid the thread migrations between processors. In this transformation, we authorize the parallel threads to be stretched up to the maximum possible thread-capacity of a processor, which can be at most the deadline of the parallel task. Let x denote the thread-capacity of a particular processor (all identical processors have the same value), which is calculated as $x = \lfloor \frac{D_i}{C_{i,j}} \rfloor$. This means that

[4] A constrained deadline real-time task has a deadline no more than its period.

each processor can contain at most x complete threads executing sequentially. The result of the transformation is a set of $\lfloor \frac{C_i}{x} \rfloor$ implicit-deadline threads each has a WCET equals to $(x * C_{i,j})$. Also, at most on implicit-deadline thread which has a WCET equals to $(C_i - ((x * C_{i,j}) * \lfloor \frac{C_i}{x} \rfloor))$. The resulted threads are scheduled using gEDF on m identical multiprocessors.

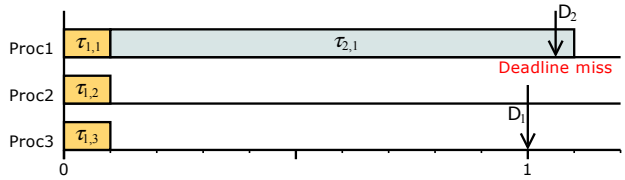
As shown in the example in Figure 1, parallel task τ_i from Inset 1(a) is transformed using the Partially-Stretched Transformation, and the result is shown in Inset 1(b). The processor capacity of task τ_i is equal to 2, and the result of transformation is 4 sequential implicit-deadline threads characterized by $(8, 11, 11)$. It is clear that the Partially-Stretched transformation does not force any threads to migrate prior to the scheduling process, and this is the advantage of this transformation over the Fully-Stretched Transformation.

The worst-case execution time of a task is a pessimist value, and usually the jobs do not execute up to this value. In the case of a global work-conserving scheduling algorithm, when a job finishes its execution earlier than expected by the WCET, then the scheduler will not allow to leave a processor idle while there are active jobs ready to execute, and it will allow an active job to be released earlier. In the case of partitioned scheduling, the scheduling algorithm assigns tasks to processors, and then migration between processors is not allowed. Hence, if a job finishes its execution earlier than expected and there are no active jobs waiting on this processor, the processor will stay idle even of there are active jobs waiting on the other processors.

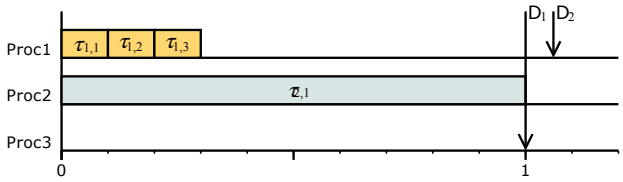
Also, it had been proved in [8] that fixed job priority algorithms (which include gEDF) are predictable, *i.e.* a schedulable taskset is guaranteed to stay schedulable when one or more of its tasks execute for less than its worst-case execution time. This property is another advantage of global scheduling over partitioned scheduling. We can conclude that the Parallel and Partially-Stretched scenarios, which use gEDF to schedule all the threads of the taskset, behave better than the Fully-Stretched scenario (which uses partitioned scheduling for most of the executed tasks) in the case of lower execution time of jobs. Hence, the processors will be efficiently used.

Advantage of stretching over parallelism in real-time systems:
The scheduling of real-time tasksets on multiprocessor systems is more complicated than the uniprocessor systems. A famous problem had been shown in [9] called the *Dhall effect*, in which a low utilization taskset can be non-schedulable regardless of the number of processors in the platform. Later, it had been proved in [10] that this problem happens when a low utilization taskset contains a high utilization task. We will show using an example, that this problem happens in the case of the Parallel Scenario, while the stretching scenarios solves it. The used example is inspired from [10].

Let us consider a taskset τ that consists of 2 tasks that executes on 3 unit-speed processors ($m = 3$). The first task τ_1 has a deadline equal to 1, and it consists of 3 threads each has a WCET equals to 2ϵ, where ϵ is slightly greater



(a) Parallel Scenario: All the threads of the task τ_1, which have the highest priority according to gEDF, execute on all 3 processors. While the highest utilization task τ_2 is delayed and hence it misses its deadline.



(b) Stretched Scenario: Parallel threads of task τ_1 are executed as sequentially as possible. Then, task τ_2 have the chance to execute at time 0.

Fig. 2. An example shows how stretching a parallel task helps in solving the Dhall effect problem.

than zero. The second task τ_2 has a deadline equals to $1+\epsilon$, and it has a single thread with WCET equals to 1. The utilization of each task is calculated as $U_1 = 2m\epsilon$ and $U_2 = \frac{1}{1+\epsilon}$. Hence, the total system's utilization approaches to 1 since $\epsilon \to 0$. The taskset is shown in Figure 2.

When gEDF is used, at time $t = 0$, the first job of task τ_1 has a higher priority than the job of task τ_2, because it has an earlier absolute deadline. All the threads of τ_1 have the same priority of τ_1, and according to the Parallel Scenario, they will execute in parallel and they will occupy all the available processors in the systems. At time $t = 2\epsilon$, the threads finish their execution, and then task τ_2 can start its own. Unfortunately, task τ_2 misses its deadline. The scenario is shown in Inset 2(a).

However, when a stretching scenario is used (either fully-stretched or partially-stretched transformation has the same result), the parallel threads of the low utilization task will be forced to execute sequentially and hence they will occupy lower number of processors. As a result, the higher utilization task (τ_2 in the example) will start its execution earlier and it will respect its deadline. Inset 2(b) shows the effect of stretching on the Dhall effect problem, and how it solved it.

3 The effect of parallelism on the tardiness of Soft real-time systems

In soft real-time systems, a deadline miss during the scheduling process of a taskset does not have catastrophic effects on the correctness of the system as in hard real-time systems. However, a deadline miss of a task will reduce the quality of service (QoS) provided by the application. So, in order to keep the QoS at an acceptable rate and better analyze it, it is necessary to determine an upper bound of tardiness for each task in a system regarding a specific scheduling algorithm.

A tardiness of a job in real-time systems is defined as its delay time, *i.e.* the time difference between the deadline and the actual finish time of this job. For a real-time task that generates a number of jobs, the tardiness of one job has a cascaded effect on the successor jobs, since the next jobs have to wait for the delayed one to finish its execution before they can start their own execution, which means that they will be delayed as well. The tardiness of a task is defined as the maximum tardiness among its generated jobs.

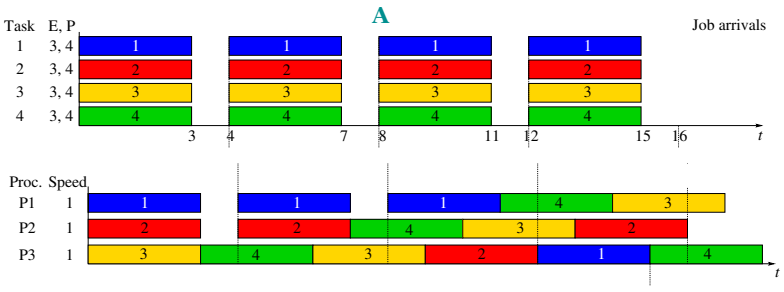


Fig. 3. An example (from [11]) shows the tardiness of tasks during the scheduling of a taskset consists of 4 tasks on 3 identical processors using gEDF.

Few researches have been done regarding the problem of identifying an upper bound to the tardiness of sporadic sequential implicit-deadline tasks on multiprocessors when global EDF is used, such as in [11][5]. In [13], an upper bound of tardiness under preemptive and non-preemptive global EDF is proved. In this work, we intend to present the latter tardiness bound in the case of preemptive gEDF and discuss its usefulness in the calculations of tardiness in the case of parallel tasks. In the future, we aim at calculating an adapted upper bound for parallel tasks.

For the sake of explaining the upper bound of tardiness from [13], Consider that τ is a sporadic taskset that is scheduled using gEDF on m identical proces-

[5] In [12], the authors mentioned that a proof published in [11] contains an error. Until now, we are not aware of the availability of a correction for this error.

sors. Each task $\tau_k \in \tau$ is a sequential implicit-deadline task that has a WCET C_k and a deadline D_k(equal to minimum arrival time or period). The upper bound of tardiness of each task in τ is given in the following theorem:

Theorem 1 (from [13]). *Global EDF (gEDF) ensures a tardiness bound of*

$$\frac{\sum_{i=1}^{\Lambda} \varepsilon_i - C_{min}}{m - \sum_{i=1}^{\Lambda-1} \mu_i} + C_k \tag{1}$$

to every sequential task τ_k of a sporadic implicit-deadline task system τ with $U_{sum} \leq m$, where:
ε_i (resp. μ_i) denotes the i^{th} execution cost (resp. task utilization) in non-increasing order of the execution costs (resp. utilization) of all the tasks.

$$\Lambda = \begin{cases} U_{sum} - 1, & U_{sum} \textit{is integral} \\ \lfloor U_{sum} \rfloor, & \textit{otherwise} \end{cases} \tag{2}$$

The detailed proof of Theorem 1 can be found in [13]. The used approach is based on comparing the processors allocations using gEDF with a concrete task system in a processor sharing [6].

As we can notice from Equation 1, the tardiness is calculated as the worst-case execution time of each task plus a fixed value w.r.t. the global parameters of the taskset. By analyzing this bound, we conclude that the tardiness of a taskset is reduced if, at least, one of the following is applied:

- a decrease in the value of the highest WCET and utilization of the tasks in the set (ε and μ),
- an increase in the WCET of tasks in τ (C_{min}),
- an increase in the number of processors,
- a decrease in the total utilization of the taskset (U_{sum}) which affects the value of Λ from Equation 2.

In our parallelism model described in Section 2, a thread in a parallel task is considered as a sequential task. Hence, we can apply theorem 1 on each thread $\tau_{i,j} \in \tau_i \in \tau$ individually. The tardiness of a job of a parallel task is the maximum tardiness among its threads, and then the tardiness of a parallel task is the maximum tardiness among its jobs.

It is worth noticing that the upper bound of tardiness is computed for sporadic implicit-deadline tasks, while the fully-stretched transformation generates at most two threads that scheduled using gEDF and may be delayed, which are

[6] A processor sharing is an idle fluid schedule in which each task executes at a precisely uniform rate given by its utilization (from [13]).

τ'_{cd} and τ'_{imp}. As a solution to this problem, and in order to use the tardiness bound from Theorem 1, we propose a modification to the constrained deadline threads τ'_{cd} to be be executing completely using gEDF, and it is converted to implicit-deadline threads. An example of this modification is shown in the example in Figure 1(d).

Let the tardiness of a parallel task $\tau_k \in \tau$ be denoted by x_k when it executes in the parallel scenario described above (all threads are activated in parallel). If task τ_k is partially stretched, then the threads of the parallel tasks will be stretched which will increase the utilization and execution time of threads. On another hand, the number of threads to be scheduled using gEDF is reduced on m processors. So, after the partially-stretched scenario, the values of C_{min}, ε and μ will increase. Also, when task τ_k is fully-stretched, the resulted fully-stretched threads (their utilization equals to 1) will be assigned dedicated processors, and at most 2 threads from each parallel task will be scheduled using gEDF only. As a result, the total utilization of the taskset τ will be reduced and also the number of processors on which gEDF algorithm is used, in addition to the effects of the partially-stretched scenario. The stretching scenarios have the advantage of reducing the number of threads that may cause a tardiness due to deadline misses when compared to the parallel scenario. Hence, the tardiness of the parallel tasks will be reduced as a result.

Based on these effects, we can conclude that the tardiness bound of parallel tasks is not comparable with the bound after stretching. Because stretching scenarios change the taskset in a way that can increase and decrease the tardiness bound at the same time. Hence, the theoretical tardiness bound of Theorem 1 cannot determine the performance of parallel and stretched tasks in the scheduling problem using gEDF, and it cannot be used as an indication to the performance of parallelism in real-time systems. As a result, we will use experimental analysis to simulate the scheduling of the different scenarios of parallel task execution and to give us an indication on the performance.

4 Experimental Analysis

In this section, we show the simulation results of the experiments conducted using randomly-generated tasksets to evaluate the performance of parallel execution in comparison with the stretching execution scenarios described in Section 2. The results are obtained by simulating the scheduling of a large number of parallel tasksets with different utilization on a platform of 16 identical processors when global EDF is used.

The simulation process is based on an event-triggered scheduling. This means that at each event in the interval $[0, 3 * H)$, where H denotes the hyper period of the scheduled taskset τ and is defined as the least common multiple of periods, the scheduler is awakened and it decides which jobs have the highest priorities to execute on the available processors. According to EDF, the job with the earliest absolute deadline has the highest priority. We consider that a parallel job is blocked either by the execution of a higher priority thread or by an earlier job

that has been delayed. During the simulation process, we calculate the tardiness of each job of parallel tasks, in order to calculate the average tardiness of tasks in the set, while varying three parameters: the execution behavior of parallel tasks (either Parallel "Par", Fully-Stretched "F-Str" or Partially-Stretched "P-Str"), the utilization of tasks within each taskset (either High "Hi" or Low "Lo"), this is done by varying the number of tasks of each taskset, and finally the number of parallel threads within each parallel task (maximum of 3 or 10 threads/task).

We used a simulation tool called YARTISS [14], which is a multiprocessor real-time scheduling simulator developed by our research team. It contains many scheduling algorithms and task models (including parallel tasks), and it can be used easily for both hard and soft real-time systems.

For each system utilization from 1 to 16, we generated $50,000$ tasksets randomly. The number of parallel tasks within each taskset is varied from 3 to 12 tasks/taskset. This variation affects the structure of tasks because, for a fixed utilization, increasing the number of tasks means that the taskset's utilization will be distributed on a larger number of tasks, which will lower the average utilization of tasks within the taskset. Also, we can control the percentage of parallelism within a taskset by varying the number of parallel threads of each task during task generation. This can help in analyzing the effect of parallelism on scheduling as we will see below.

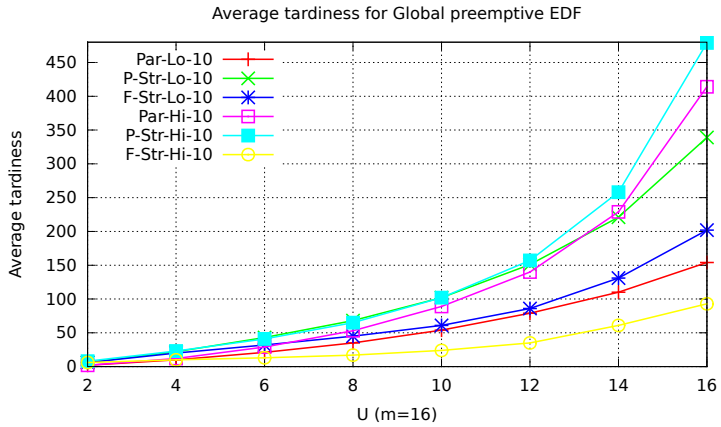
Regarding the generation of parallel tasksets, our task generator is based on the Uunifast-Discard algorithm [15] for random generation of tasks. This algorithm is proposed by Davis and Burns to generate randomly a set of tasks of a certain total utilization on multiprocessor systems. The number of tasks and their utilization are inputs of this algorithm. The taskset generator is described briefly as follows:

- The algorithm takes two parameters n and U, where n is the number of parallel tasks in the set and U is the total utilization of the taskset ($U > 0$).
- The Uunifast-Discard algorithm distributes the total utilization on the taskset. A parallel task τ_i can have a utilization U_i greater than 1 which means that its threads cannot be stretched completely, and it has to execute in parallel.
- The number of threads and their WCET of each parallel tasks are generated randomly based on the utilization of the tasks. The maximum number of threads is fixed to be either 3 or 10 threads per parallel task.

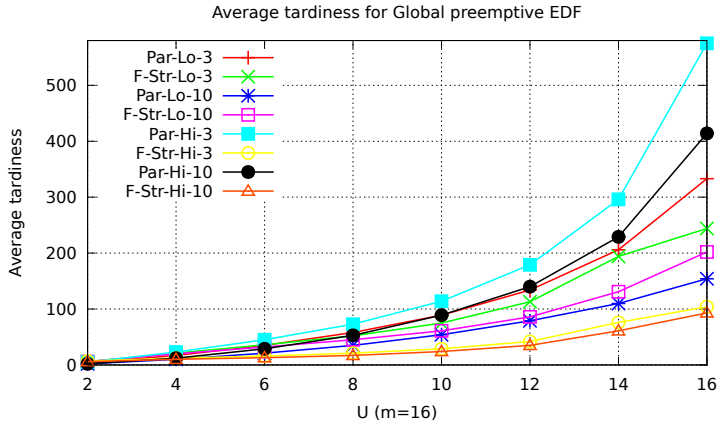
In order to limit the simulation interval and reduce the time needed to perform the simulation, which is based on the length of the hyper period of each taskset, we used the limitation method proposed in [16], which relays on using a considerate choice of periods of the tasks while generation so as to reduce their least common multiple. Using this method, we implemented our task generator to choose periods of tasks in the interval $[1, 25200]$.

Analysis of experimental results

For each taskset, the average of the maximum tardiness of all tasks is computed. The results are showed in Figure 4, which consists of 2 insets. Inset 4(a) shows



(a) Comparison of the average tardiness of taskset when executing on $m = 16$ using all execution scenarios for parallel tasks.



(b) Comparison of the average tardiness of taskset when executing on $m = 16$ focusing on the parallel and the Full-stretch execution scenarios.

Fig. 4. Simulation results of average tardiness of parallel tasks by varying their execution behavior (Parallel "Par", Full-stretch "F-Str" and Partially-stretch "P-Str"), the utilization of tasks within tasksets (high "Hi" and Low "Lo") and the maximum number of threads within each parallel task (either 3 or 10 threads/task).

the comparison of average tardiness of tasksets for the three types of execution behaviors, while Inset 4(b) focuses on the comparison betzeen the parallel and the fully-stretched execution behavior.

The average tardiness is calculated by considering the schedulable tasks (their tardiness equal to zero), so as to give us an indication on the number of deadline misses happened during simulation. Moreover, the x-axis of the insets of Figure

4 represents the utilization of the scheduled tasksets (from 1 to 16 on a system of 16 identical processors), while the y-axis represents the average tardiness of the tasksets.

The rest of this section discusses the results in details.

The effect of utilization of tasksets on tardiness.

Referring to Figure 4, most of the tasksets have negligible tardiness when tasksets have utilization less than 4. Then the tardiness increases differently (based on the used scenario) for higher utilization. These results are quite logical, since the number of processors in the simulation is always considered to be $m = 16$. Hence, lower utilization tasksets mean lower execution demand from processors which increases schedulability and reduces deadline misses. Starting from $U_{taskset} > 4$, we can notice an increase in tardiness for the average tardiness values which varies based on the execution scenario.

However, we can conclude by looking at these results, and specially when $U_{taskset} = 16$, that the partially-stretched transformation has always the highest tardiness values when it is compared with the other two execution scenarios (parallel and fully-stretched) with the same parameters. As a result, the partially-stretched scenario can be seen as the least appealing execution scenario from the schedulability point of view. This is expected since the partially-stretched transformation delays the execution of part of the parallel threads of a task so as to execute sequentially. This is done even if there are available processors for them to execute earlier than the defined activation time specified by the transformation. In Figure 4(a), the partially-stretched scenario (P-Str) has the highest tardiness in both groups (high and low utilization) when the number of threads of each parallel task has a maximum of 10 threads.

The performance of the other two scenarios (parallel and fully-stretched) is not comparable at this point, and they are affected by the utilization of parallel tasks and the number of threads in each task, and this will be discussed in the next paragraph.

The effect of the number of tasks per taskset on tardiness.

The difference between the parallel and the fully-stretched execution scenarios of parallel tasks is shown in Figure 4(b). Since the Uunifast-Discard algorithm that we used for generating tasksets divides the total utilization of taskset on the number of tasks per taskset, varying the number of tasks while fixing the system's utilization will vary the utilization assigned for each task. In our parallel task generator, the maximum possible number of threads depends on the total execution time of the task (respectively, its utilization). So, lowering the task utilization means a lower execution time which increases the probability of generating tasks with low number of parallel threads. We can notice that the highest tardiness of tasksets is caused from the high-utilization tasksets executed using the parallel execution scenario (Par-Hi in Figure 4(b)), while their respective tasksets executing using the fully-stretched scenario (F-Str in Figure 4(b))

have the lowest tardiness, regardless of the number of parallel threads within each task.

Regarding the low-utilization tasks, we can notice that the tardiness of the tasksets depends on the number of parallel threads within each task. As shown in Figure 4(b), the fully-stretched tasksets whose tasks consist of 3 threads (F-Str-Lo-3) have lower but relatively close tardiness than the parallel scenario (Par-Lo-3). However, the parallel executing tasksets whose tasks consist of 10 threads (Par-Lo-10) are behaving better than the fully-stretched scenario (F-Str-Lo-10).

In this case, the scheduling of the parallel scenario is clearly better than the fully-stretched scenario as shown in Figure 4(b). This can be explained by noticing that in this case the number of parallel threads in a task is high while its utilization is low. Since the threads inherit the deadline and period of their original task, then the threads have low utilization as well. The parallel scenario gives the scheduling algorithm more freedom in scheduling the parallel threads, by activating them all at the same time, and choosing their execution order based on the availability of processors. While the fully-stretched scenario forces the parallel tasks to execute sequentially even if this approach is not work-conserving and it might cause delays in the scheduling process.

From the results conducted by simulation, we have now an experimental indication on the effect of parallelism on the scheduling of real-time systems. It is possible now to overrule the typical assumption that parallelism has always negative effects on scheduling. As we have shown above, the parallel scheduling is better than its sequential alternatives when tasks have low number of parallel threads. According to this, the scheduler can get better scheduling decisions while parallelizing the execution of certain tasks on multiple processors than the sequential execution. This result matches the motivation of parallelism and its practical uses in non-real time systems.

5 Conclusion

In this paper, we were interested in studying the effect of parallelism in real-time systems. The problem is summarized as the scheduling of sporadic implicit-deadline parallel tasks on multiprocessors using global earliest deadline first (gEDF) as scheduling algorithm. The parallel tasks are either executed in a parallel scenario in which all the threads of the parallel tasks execute in parallel as soon as possible, or in a stretching scenario, in which the threads are executed as sequentially as possible. We proposed two stretching scenarios based on the number of thread migrations and preemptions required by the transformation: partially and fully stretched. In the latter, threads are stretched to form transformed threads with utilization equal to 1, but it requires higher number of migrations and preemptions between processors and jobs.

Using extensive simulation, we showed that parallelism did not cause major negative effects on the scheduling of real-time systems. Admitting that sequential execution of tasks has better results in general than parallelism, There are certain cases where parallelism behaves better and has lower tardiness values

than stretching. Based on these remarks and results, we can overrule the common assumption in real-time systems against parallelism, and that tries to avoid parallel structure in order to get better scheduling results.

In the future, we aim at extending our work, and provide theoretical analyses to support our experimental results, by providing an upper bound of tardiness adapted to parallel real-time tasks on multiprocessor systems. Also, we are looking forward to analyze scheduling algorithms other than gEDF algorithm that we used in this paper. Based on this, we can classify the common scheduling algorithms in real-time systems based on their ability to schedule parallel tasks with low tardiness bounds.

Finally, we aim at generalizing our task model of parallel tasks, so as to include more complicated structures of parallel threads, such as the multi-threaded segment model and the Directed Acyclic Graphs. Such task models are used to represent practical parallel programming APIs.

References

1. "Mixed criticality systems," European Commission Workshop on Mixed Criticality Systems, Brussels, Belgium, February 2012.
2. "Posix threads programming." [Online]. Available: https://computing.llnl.gov/tutorials/pthreads/
3. "OpenMP." [Online]. Available: http://www.openmp.org
4. R. I. Davis and A. Burns, "A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems," *ACM Computing surveys*, pp. 1 – 44, 2011.
5. A. Saifullah, D. Ferry, K. Agrawal, C. Lu, and C. Gill, "Real-Time Scheduling of Parallel Tasks under a General DAG Model," Washington University in St Louis, Tech. Rep., 2012.
6. S. K. Baruah, V. Bonifaciy, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *Proceedings of the 33rd IEEE Real-Time Systems Symposium (RTSS)*, Dec. 2012, pp. 63–72.
7. K. Lakshmanan, S. Kato, and R. (Raj) Rajkumar, "Scheduling Parallel Real-Time Tasks on Multi-core Processors," in *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, 2010, pp. 259–268.
8. R. Ha and J. Liu, "Validating timing constraints in multiprocessor and distributed real-time systems," in *14th International Conference on Distributed Computing Systems*. IEEE Comput. Soc. Press, 1994, pp. 162–171.
9. S. K. Dhall and C. L. Liu, "On a Real-Time Scheduling Problem," *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978.
10. C. A. Phillips, C. Stein, E. Torng, and J. Wein, "Optimal time-critical scheduling via resource augmentation (extended abstract)," in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '97, 1997, pp. 140–149.
11. P. Valente and G. Lipari, "An upper bound to the lateness of soft real-time tasks scheduled by edf on multiprocessors," in *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, 2005, pp. 10 pp.–320.
12. ——, "An upper bound to the lateness of soft real-time tasks scheduled by EDF on multiprocessors," Scuola Superiore S.Anna, Tech. Rep. RETIS TR05-01, 2005.

13. U. Devi, "Soft Real-Time Scheduling on Multiprocessors," Ph.D. dissertation, University of North Carolina at Chapel Hill, Chapel Hill, Sweden, 2006.
14. "YaRTISS simulation tool." [Online]. Available: http://yartiss.univ-mlv.fr/
15. R. Davis and A. Burns, "Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," *Real-Time Systems*, vol. 47, no. 1, pp. 1–40, 2011.
16. J. Goossens and C. Macq, "Limitation of the Hyper-Period in Real-Time Periodic Task Set Generation," in *Proceedings of the 9th International Conference on Real-Time Systems (RTS)*, Mar. 2001, pp. 133–148.