Response-Time Analysis of Synchronous Parallel Tasks in Multiprocessor Systems

Cláudio Maia
CISTER/INESC TEC, ISEP
Porto, Portugal
crrm@isep.ipp.pt

Marko Bertogna
University of Modena
Modena, Italy
marko.bertogna@unimore.it

Luís Nogueira
CISTER/INESC TEC, ISEP
Porto, Portugal
lmn@isep.ipp.pt

Luis Miguel Pinho
CISTER/INESC TEC, ISEP
Porto, Portugal
lmp@isep.ipp.pt

ABSTRACT

Programmers resort to user-level parallel frameworks in order to exploit the parallelism provided by multiprocessor platforms. While such general frameworks do not support the stringent timing requirements of real-time systems, they offer a useful model of computation based on the standard fork/join, for which the analysis of timing properties makes sense. Very few works analyse the schedulability of synchronous parallel real-time tasks, which is a generalisation of the standard fork/join model.

This paper proposes to narrow the gap by presenting a model that analyses the response-time of synchronous parallel real-time tasks. The model under consideration targets tasks with fixed priorities, composed of several segments with an arbitrary number of parallel and independent units of execution.

We contribute to the state-of-the-art by analysing the response-time behaviour of synchronous parallel tasks. To accomplish this, we take into account concepts previously proposed in the literature and define new concepts such as carry-out decomposition and sliding window technique in order to compute the worst-case workload in a window of interest. Results show that the proposed approach is significantly better than current approaches, improving the state-of-the-art analysis of parallel real-time tasks.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*Scheduling*

General Terms

Design, Algorithms, Theory

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
RTNS 2014 , October 8 - 10 2014, Versailles, France
Copyright 2014 ACM 978-1-4503-2727-5/14/10 ...$15.00.
http://dx.doi.org/10.1145/2659787.2659815.

Keywords

Parallel Task Model, Job-level Parallelism, Real-Time

1. INTRODUCTION

The real-time systems domain is currently facing the challenge of how to exploit the immense computing power offered by the next-generation of many-core systems for time-critical applications [1]. To this extent, several techniques have already been proposed for scheduling real-time tasks in multiprocessor systems [2]. However, the latest market requirements and technology advancements require both predictability and performance from real-time applications.

In order to explore the inherently parallel computing power available, new real-time computing models have recently been proposed with a special focus on *job-level parallelism* or *intra-task parallelism* [3, 4]. For example, in the synchronous parallel task model proposed in [4] (depicted in Figure 1), real-time jobs are composed of consecutive segments, each containing a different number of independent sub-jobs. Different names are used in the literature to name these independent sub-jobs, i.e, thread, sub-task, parallel job or, in short, *p-job*. Segments have precedence constraints among themselves, meaning that p-jobs belonging to a certain segment can only start their execution after all the p-jobs in the previous segment have been completed. Moreover, there is no restriction on the number of segments per job, neither on the number of p-jobs executing in a segment. Such a model is a generalisation of the fork/join model presented in [3] to model Java Fork/Join [5] or OpenMP [6] constructs.

This paper focuses on the schedulability analysis of synchronous parallel tasks in multiprocessor systems composed of identical processors, where tasks have a fixed priority and the structure of each task is known *a priori*. Our approach improves over the work reported in [7], providing tighter schedulability conditions and extending the analysis to fixed-priority task systems. Tighter upper-bounds on the workload within a window of interest are derived by computing the response-time upper bounds of the interfering jobs, similarly to the technique proposed in [8] for sequential task sets. However, there are subtle differences to consider when dealing with parallel tasks. The analysis in [8] cannot be applied as it is, but needs to be properly modified to take into account the different task structure in a predictable way.

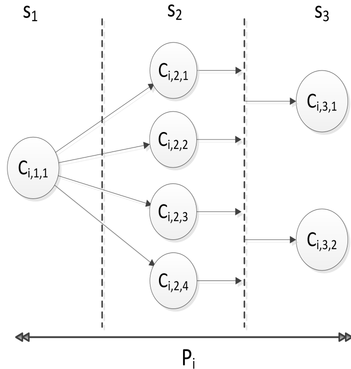


Figure 1: Example of a synchronous parallel task τ_i composed of three segments, with one, four and two p-jobs, respectively.

Contributions: Our contributions are the following:

- We present a schedulability analysis for synchronous parallel sporadic tasks scheduled with global Fixed Priority on identical multiprocessors. Existing schedulability analyses are based on global Earliest Deadline First [7], or consider partitioned approaches [9].
- We highlight the main issues in the response-time analysis of parallel tasks for deriving predictable upper bounds on the interfering contributions in a window of interest.
- We provide a pseudo-polynomial algorithm to compute an upper bound on the response time of each parallel task.
- We present an extensive set of experiments showing that the proposed approach outperforms the state-of-the-art schedulability techniques for parallel real-time task systems.

The remainder of this paper is organised as follows. Section II presents the state-of-the-art of parallel real-time tasks. Section III describes the system model. Section IV introduces some preliminary results based on the notion of critical interference. Using these results, Section V details the schedulability analysis for fixed-priority synchronous parallel tasks. Section VI presents the simulation results obtained from experiments conducted using synthetically generated task sets. Finally, section VII concludes the paper and presents the future work.

2. RELATED WORK

Goossens and Berten [10] redefined a classification for different types of parallel tasks. In this classification, a job may be classified as rigid, moldable or malleable. In a rigid job, the number of processors assigned to each segment is fixed and determined *a priori*, so that either all p-jobs of a segment are executed, or none of them is scheduled for execution. In a moldable job, the number of processors assigned to each segment is determined by the scheduler, but cannot change once the segment starts executing. Finally, a malleable job allows the number of processors assigned to each segment to change dynamically.

The problem of multiprocessor scheduling of parallel real-time tasks was covered by Han and Lin in [11] where the authors prove that the problem of scheduling fixed-priority parallel jobs is NP-Hard. Drozdowski [12] focuses on the problem of scheduling parallel tasks with the objective of minimizing the makespan.

Concerning rigid tasks, Goossens and Berten [10] proposed a scheduling algorithm for parallel rigid real-time tasks based on gang scheduling. Moldable tasks were studied by Manimaran et al. [13] where they proposed a non-preemptive Earliest Deadline First (EDF) approach that considers parallel real-time tasks. Kato and Ishikawa [14] proposed the Gang-EDF algorithm, which applies EDF to the traditional gang scheduling scheme.

Jansen [15], Collette et al. [16], and Korsgaard and Hendseth [17] studied malleable tasks. Jansen [15] focused on minimizing the makespan but without considering real-time constraints. Collette et al. [16] studied the problem of global scheduling of sporadic task systems on multiprocessors considering job-level parallelism. Korsgaard and Hendseth [17] proposed a schedulability test for malleable tasks scheduled with global EDF.

Lakshmanan et al. [3] study the scheduling of periodic fork/join real-time tasks on multiprocessor platforms. Each task is divided into sequential and parallel segments. Parallel segments must be preceded and followed by a sequential segment. All parallel segments must have the same number of threads, and the number of threads cannot be greater than the number of processors in the platform. In order to schedule such tasks in a multiprocessor platform, the authors propose the decomposition of fork/join tasks by applying a task stretch transform algorithm. Moreover, a resource augmentation bound is derived for the decomposed task set when partitioned deadline monotonic scheduling is used.

Saifullah et al. [4] generalise the fork/join model presented in [3], named *synchronous parallel* task model. In this model, parallel tasks may have any number of segments, and the number of parallel threads within any segment can be greater than the number of cores in the platform. In [4], the decomposition of periodic parallel tasks into constrained-deadline sequential tasks is proposed. For the decomposed task sets, a resource augmentation bound is derived for global EDF and partitioned deadline monotonic scheduling policies.

A schedulability condition for synchronous parallel tasks scheduled with global EDF has been presented in [7], extending the traditional interference-based analysis for serial task models. The authors introduce the concept of *critical interference* to capture the interference of parallel threads. In this paper, we will borrow the concept of critical interference to propose tighter schedulability conditions based on the response-time analysis of synchronous parallel tasks.

In [9], a worst-case response-time analysis is presented for fork/join tasks under partitioned fixed-priority scheduling. The analysis iterates over the segments of a fork/join task, selecting the worst-case response time of each segment. The authors show that fork/join tasks may present a larger worst-case response time due to the interference of sequential tasks.

In [18], the Fork/Join OS (FJOS) is presented, an operating system based on Composite OS, comparing its behaviour with the GOMP implementation on Linux. Moreover, the schedulability analysis technique proposed in [9] is adapted to include overheads based on real measurements in FJOS. As in [9], this approach is also based on partitioned fixed-priority scheduling.

3. SYSTEM MODEL

We consider the problem of scheduling synchronous parallel real-time tasks with fixed priority on a system composed of m identical processors with uniform memory access. In our model, each task releases a sequence of jobs where each job instance is allowed to execute in more than one core at the same instant. Without loss of generality, all time intervals and task parameters are assumed to be integer multiples of the system clock.

Let $\tau = \{\tau_1, ..., \tau_n\}$ denote the set of n sporadic tasks. Assume tasks are indexed in priority order, task τ_1 being the highest priority one. Each task τ_i in the task set τ releases an infinite sequence of jobs separated by at least T_i time units. Each task has a deadline $D_i \leq T_i$ (i.e., constrained deadline model), meaning that each of its jobs needs to complete its execution at most D_i time units after its release. Moreover, each task τ_i is characterised by a sequence of segments $s_i = \{\sigma_{i,1}, ..., \sigma_{i,s_i}\}$, where each segment $\sigma_{i,j}$ is composed of a set of $m_{i,j}$ parallel jobs (or in short p-jobs), $\{J_{i,j,1}, ..., J_{i,j,m_{i,j}}\}$, each one having the same priority of the task that spawns it. The parallel jobs are independent sequential threads that can be executed in parallel, i.e., in different processors at the same time instant (see Figure 1). Before a segment starts executing any of its p-jobs, all the p-jobs of the preceding segment (if any) must have been completed. Besides these precedence constraints, parallel jobs are independent, and there are no other shared resources than processing units.

To clarify, $m_{i,j}$ denotes the number of p-jobs within the j-th segment of task τ_i. In this work, we allow the number of p-jobs of a segment to be greater than the number of cores, so that $m_{i,j}$ may be greater than m for some segment $\sigma_{i,j}$. The maximum degree of parallelism of a task is denoted as m_i and is defined as $m_i = \max_j\{m_{i,j}\}$.

A fully preemptive system is assumed where any executing p-job may be preempted and resumed later without any cost. At any given instant, the m ready p-jobs with the highest priority are the ones executing in the cores. Ties are broken arbitrarily.

Each p-job instance $J_{i,j,k}$ is characterized by a worst-case execution time $C_{i,j,k}$. The worst-case execution time of each segment $\sigma_{i,j}$ is given by:

$$C_{i,j} = \sum_{k=1}^{m_{i,j}} C_{i,j,k}. \tag{1}$$

The overall worst-case execution time of a task τ_i is then defined as:

$$C_i = \sum_{j=1}^{s_i} C_{i,j}. \tag{2}$$

The above equations represent the time it takes to execute a segment (Equation 1) or a task (Equation 2) in a dedicated single processor platform, i.e., with no parallelism.

The *minimum worst-case execution time* P_i of a task τ_i is the time task τ_i requires to execute when the number of processing units m is infinite, i.e., the critical path length. Formally, P_i is defined as:

$$P_i = \sum_{j=1}^{s_i} P_{i,j}, \tag{3}$$

where $P_{i,j}$ represents the worst-case execution time of the largest p-job(s) of segment $\sigma_{i,j}$. Formally,

$$P_{i,j} = \max_{k=1}^{m_{i,j}} \{C_{i,j,k}\}. \tag{4}$$

The *utilisation* U_i of task τ_i is the ratio between the task's overall worst-case execution time and period, $U_i = \frac{C_i}{T_i}$. For the task set τ, the *total utilisation* is defined as $U(\tau) = \sum_{i=1}^{n} U_i$.

For implicit-deadline sequential task sets, a necessary and sufficient condition for feasibility is $U(\tau) \leq m$ ([19]). Nevertheless, for fork/join tasks this condition is only necessary [3], as it is for the synchronous parallel task model adopted in this paper. Moreover, another necessary condition for schedulability of synchronous parallel tasks is $P_i \leq D_i$.

It is important to note that with the synchronous task model there may be feasible task sets in which some task has a utilisation larger than 100%. Serialisation techniques are not possible with such tasks, as the derived sequential task would be clearly unschedulable.

The worst-case response-time of τ_i, denoted as R_i, is given by the maximum amount of time that elapses between the release time (r_i) of any job of τ_i and its completion time. For parallel tasks, R_i clearly depends on several factors such as the inter-task and intra-task interferences; precedence constraints between the segments of the task itself; the degree of parallelism of each region; and the number of cores in the hardware platform. As it may be extremely difficult to derive the exact worst-case response time of a task in the addressed setting, a typical approach in the real-time literature is to compute an upper bound R_i^{ub} on the response-time of task τ_i.

Table 1 presents a summary of the important notation defined and used throughout the paper for quick reference and clarity.

4. CRITICAL INTERFERENCE OF PARALLEL TASKS

We propose a global fixed-priority approach for synchronous parallel tasks. In this approach, all the ready p-jobs are inserted in a global queue, from where m processors pick the m highest priority p-jobs. The proposed approach considers for each job its worst-case execution time and the interference that it suffers. If the system is schedulable, the interference is bounded and the job execution time plus the imposed interference is always less than or equal to the job's deadline.

Interference is an important concept widely used in real-time systems. For traditional sequential task sets, the interference a task τ_k suffers over an interval of length L, denoted as $I_k(L)$, is defined as the sum of all intervals of time in which τ_k is ready to execute but it cannot execute due to the execution of other higher priority tasks in the system. In particular, the interference of a higher priority task τ_i over task τ_k in an interval of length L is denoted as $I_{i,k}(L)$, and is defined as the sum of all time-intervals in which τ_i is executing but τ_k is not, even though it is ready to execute. Intuitively, the interference that a task suffers cannot be greater than the total amount of workload of the higher priority jobs.

When dealing with synchronous parallel tasks two types of interference may occur: *inter-task* and *intra-task* inter-

Table 1: Summary of notation

Symbol	Description
m	Number of processors in the platform
n	Number of tasks in the task set
τ	Set of periodic or sporadic tasks
U_i	Utilisation of task τ_i, i.e., $\frac{C_i}{T_i}$
$U(\tau)$	Total utilisation of the task set τ
T_i	Period of task τ_i
D_i	Relative Deadline of task τ_i
C_i	Overall worst-case execution time requirement of τ_i
P_i	Minimum worst-case execution time of task τ_i
s_i	Number of segments in task τ_i
m_i	Maximum degree of parallelism of task τ_i
$C_{i,j}$	Overall worst-case execution time of segment $\sigma_{i,j}$
$P_{i,j}$	Minimum worst-case execution time of segment $\sigma_{i,j}$
$m_{i,j}$	Number of p-jobs within segment $\sigma_{i,j}$
$C_{i,j,k}$	Worst-case execution time of p-job $J_{i,j,k}$
r_i	Release time of a job of task τ_i
d_i	Absolute deadline of a job of task τ_i
R_i	Worst-case response time of task τ_i
R_i^{ub}	Upper-bound of R_i
L	Generic interval $[r_k, r_k + R_k^{ub}]$
$I_k(L)$	Critical interference on task τ_k in any interval of length L
$I_{i,k}(L)$	Critical interf. of τ_i on τ_k in any L
$I_{i,k}^p(L)$	Critical interf. of τ_i on τ_k with depth at least p in any L
$W_i^p(L)$	Workload of task τ_i for at least p p-jobs in the interval L

ference. Inter-task interference is the interference caused by other higher priority tasks executing in the system. This is the standard notion of interference widely used in traditional sequential models. Instead, intra-task interference is peculiar to parallel task systems, and is defined as the self-interference due to parallel jobs of the same task instance.

In order to compute the interference of a parallel task, we adopt the concept of *critical thread*[1], as previously defined in [7].

Definition 1. *A thread is* critical *if it is the last one to complete among the threads belonging to the same segment.*

For deriving the worst-case response time of a task, it is then sufficient to characterise the interference imposed to its critical threads, as they are the ones suffering the largest interference.

Definition 2. *The* critical interference $I_k(L)$ *on task* τ_k*, in any interval of length L, is defined as the cumulative time in which a* critical thread *of task* τ_k *is ready to execute but it cannot due to the execution of other parallel jobs.*

Given the above definitions, the following theorem simply follows.

[1]While we prefer using the term parallel job instead of thread, we decided here to keep the name "thread" for homogeneity with the original definition. However, both terms are interchangeably used in this paper.

Theorem 1. *Given a set of synchronous parallel tasks* τ *scheduled by any work-conserving*[2] *algorithm on m identical cores, the worst-case response-time of each task* τ_k *can be upper bounded by* R_k^{ub} *if*

$$P_k + I_k(R_k^{ub}) \leq R_k^{ub}. \quad (5)$$

Proof. Consider the job of τ_k that leads to the worst-case response time R_k. Let r_k be its release time. Within a scheduling window $[r_k, r_k + R_k^{ub}]$, Equation (5) guarantees that all s_k critical threads have sufficient time to execute P_k time-units, while accommodating the interference suffered from other threads, accounted for in $I_k(R_k^{ub})$. Since the execution requirement of each critical thread cannot exceed the minimum worst-case execution time of the corresponding segment, Equation (3) guarantees that all critical threads complete their execution within the considered interval, proving the theorem. □

The problem of the above theorem is that computing the exact interference imposed on the considered task is difficult. To sidestep this problem, a common approach is to express the total interference as a function of individual task interfering contributions, and upper bound such contributions with the worst-case workload executed by each task in the considered window.

Definition 3. *The critical interference* $I_{i,k}(L)$ *imposed by task* τ_i *on task* τ_k *in any interval of length L is defined as the cumulative workload executed by p-jobs of task* τ_i *while a critical thread of* τ_k *is ready to execute but is not executing.*

Differently from the sequential case, each task τ_i may contribute with different p-jobs at the same time to the individual interference on a task τ_k. In the particular case when $i = k$, the critical interference $I_{k,k}(L)$ may include the interfering contributions of (non critical) p-jobs of task τ_k on itself, i.e., the intra-task interference.

The next lemma allows expressing the total interference as a function of single task interferences.

Lemma 1. *For any work-conserving algorithm, the following relation holds:*

$$I_k(L) = \frac{1}{m} \sum_{\forall \tau_i} I_{i,k}(L). \quad (6)$$

Proof. From the work-conserving property of the considered scheduler, it follows that whenever a critical thread of τ_k is interfered, all m cores are busy executing other p-jobs. Therefore, the total amount of workload executed by p-jobs interfering with critical threads of τ_k within the considered window is $mI_k(L)$:

$$\sum_{\forall \tau_i} I_{i,k}(L) = mI_k(L).$$

The lemma simply follows by rephrasing the terms. □

As previously mentioned, the individual interference $I_{i,k}(L)$ accounts for all p-jobs of τ_i interfering with τ_k, including p-jobs that are executing at the same time. In order to capture how many parallel jobs of τ_i may simultaneously interfere

[2]A scheduling algorithm is said to be *work-conserving* if it never idles a core when there is a ready task waiting to be executed.

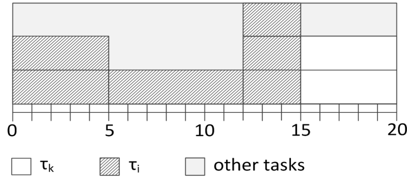


Figure 2: Task τ_i interfering on task τ_k.

with task τ_k, we will borrow from [7] the concept of *at least p-depth critical interference*[3].

Definition 4. *The* at least p-depth critical interference *of τ_i on τ_k in any interval of length L, denoted as $I^p_{i,k}(L)$, is defined as the total amount of time in which a critical thread of τ_k is ready to execute but cannot execute while there are at least p threads of task τ_i simultaneously executing in the system.*

To better understand the meaning of $I^p_{i,k}(L)$, consider the example in Figure 2, where a task τ_i interferes τ_k with two threads for five time-units, one thread for seven time-units, and three threads for three time-units. In this case, $I^1_{i,k}(L) = 15$, $I^2_{i,k}(L) = 8$, and $I^3_{i,k}(L) = 3$.

Note that by definition (Definitions 2 and 4) the following inequality holds: $I^p_{i,k}(L) \leq I_k(L)$.

The following lemma allows establishing a relation between the overall critical interference on a task τ_k and the at least p-depth critical interference of each task τ_i on τ_k.

Lemma 2. *For any work-conserving algorithm, the following relation holds:*

$$I_k(L) = \frac{1}{m} \sum_{\forall \tau_i} \sum_{p=1}^{m} I^p_{i,k}(L). \quad (7)$$

Proof. Considering each single interfering task τ_i, the amount of execution by all p-jobs of τ_i interfering with τ_k within the considered window equals $\sum_{p=1}^{m} I^p_{i,k}(L)$. The Lemma follows from Lemma 1. □

We will now extend to the parallel task model considered in this paper two results proved in [8] for sequential tasks.

Lemma 3.

$$\sum_{\forall \tau_i} \sum_{p=1}^{m} \min\left(I^p_{i,k}(L), x\right) \geq mx \quad \Leftrightarrow \quad I_k(L) \geq x.$$

Proof. *If.* We would like to prove that if $I_k(L) \geq x$, then $\sum_{\forall \tau_i} \sum_{p=1}^{m} \min\left(I^p_{i,k}(L), x\right) \geq mx$.

For a given length L, let ξ be the number of at least p-depth critical interferences $I^p_{i,k}(L) \geq x$, namely:

$$\xi = \left| \left\{ I^p_{i,k}(L) \geq x \right\}_{\forall i,p} \right|.$$

If $\xi > m$, then $\sum_{\forall \tau_i} \sum_{p=1}^{m} \min\left(I^p_{i,k}(L), x\right) \geq \xi x > mx$. Otherwise, $(m - \xi) \geq 0$, and, using Lemma 2,

$$\begin{aligned}
\sum_{\forall \tau_i} \sum_{p=1}^{m} \min\left(I^p_{i,k}(L), x\right) &= \xi x + \sum_{\forall \tau_i} \sum_{p: I^p_{i,k} < x} I^p_{i,k}(L) \\
&= \xi x + m I_k(L) - \sum_{\forall \tau_i} \sum_{p: I^p_{i,k} \geq x} I^p_{i,k}(L) && \text{[Lemma 2]} \\
&\geq \xi x + m I_k(L) - \xi I_k(L) && \left[I^p_{i,k}(L) \leq I_k(L)\right] \\
&= \xi x + (m - \xi) I_k(L) \\
&\geq \xi x + (m - \xi) x = mx. && [\text{using } I_k(L) \geq x]
\end{aligned}$$

Only if. From Lemma 2, we have

$$\begin{aligned}
I_k(L) &= \frac{1}{m} \sum_{\forall \tau_i} \sum_{p=1}^{m} I^p_{i,k}(L) \\
&\geq \frac{1}{m} \sum_{\forall \tau_i} \sum_{p=1}^{m} \min\left(I^p_{i,k}(L), x\right) \geq \frac{1}{m} mx = x.
\end{aligned}$$

□

Theorem 2. *Given a set of synchronous parallel tasks τ scheduled by any work-conserving algorithm on m identical cores, the worst-case response-time of each task τ_k can be upper bounded by R^{ub}_k if*

$$\sum_{\forall \tau_i} \sum_{p=1}^{m} \min\left(I^p_{i,k}(R^{ub}_k), R^{ub}_k - P_k + 1\right) < m(R^{ub}_k - P_k + 1)$$

Proof. If the inequality holds, Lemma 3 gives

$$I_k(R^{ub}_k) < R^{ub}_k - P_k + 1.$$

Since an integral time model is used,

$$I_k(R^{ub}_k) \leq R^{ub}_k - P_k.$$

The theorem then follows from Theorem 1. □

In the following section, the above theorem will be used to derive a sufficient schedulability test for synchronous parallel task systems scheduled with global fixed priority algorithm.

5. RESPONSE-TIME ANALYSIS

In order to exploit the theorem proved in the previous section to analyse the schedulabilty of a parallel task system, it is necessary to compute the critical interference terms. Since finding such terms is known to be a difficult problem for multiprocessor systems, a common approach is to use upper bounds that are easier to compute. An upper bound on the interference of a task τ_i in a window of length L is given by the maximum workload that τ_i can execute within the considered window. However, computing the maximum workload that can be executed by τ_i in a generic window is also a difficult task. To sidestep this problem, a typical technique is to consider pessimistic scenarios in which the workload in a given window cannot be smaller than in the worst-case situation. We hereafter describe the pessimistic scenario considered in this paper.

Consider a window of length L that spans the interval $[r_k, r_k + L]$ of a given (interfered) task τ_k. We call this interval of time the *window of interest*. Within this window, we provide an upper bound on the execution time of an interfering task τ_i. As commonly adopted in the literature, we

[3]Note that we are simplifying the analysis and notations with respect to [7], without making use of the "exact" p-depth interference, which, to our belief, is not needed for the purposes of this paper. Also the theorems presented in this section have therefore subtle differences from the corresponding ones in [7]. For instance, the case of Lemma 2, which differs from a similar result proved in [7] in that the notion of "at least p-depth critical interference" is used instead of the "exact p-depth critical interference".

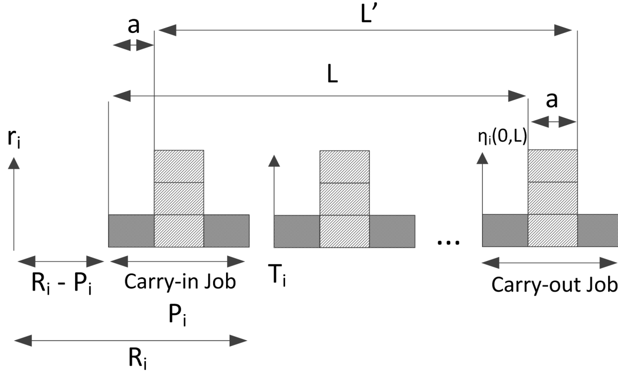


Figure 3: Densest possible packing of threads within a window of interest.

will call "carry-in job" the first instance of τ_i executing in the window of interest, having a release time before and deadline inside the window of interest. By contrast, the "carry-out job" has its release time within (or before) and deadline after the window of interest. Note that, by convention, a job that has both release time and deadline outside the window is considered to be a carry-out job. All τ_i's instances whose release time and deadline are entirely contained within the considered window will be named "body jobs".

As shown in [8], the densest possible packing of sequential jobs of a task τ_i is found when:

1. A job starts executing at the beginning of the window of interest, and completes as close as possible to its response time. In other words, the job starts executing $R_i - P_i$ time-units after its release time, in correspondence to the beginning of the window of interest.
2. All subsequent jobs of τ_i are executed as soon as possible after being released, i.e., respecting the period T_i.

Such a situation is depicted in Figure 3 for a parallel task τ_i in the window of interest.

5.1 Sliding window technique

An important observation to make is that the scenario described above may not represent the worst-case workload in the synchronous parallel task model considered in this paper. This occurs because the parallel task structure is characterized by precedence constraints that may affect the densest possible packing of p-jobs. Consider the example in Figure 3, where a task composed of three segments is considered in the above scenario. The carry-in job is fully contained inside the window of interest L, while the carry-out is only partially contained. Now, if the window of interest is shifted right by one segment (as in the window L'), the carry-in contribution decreases by one p-job, while the carry-out contribution increases by three p-jobs, leading to a larger task workload within the considered window.

In order to properly consider the worst-case workload contribution of each task in a window of interest, we check all different meaningful alignments of the window of interest with respect to the task structure. Note that shifting right the window of interest, the workload contribution has a discontinuity whenever one of the extreme points of the window coincides with a segment boundary. Therefore, we can check all possible scenarios in which the window of interest is shifted to the right from the original configuration, such that either (i) the window starts at the beginning of a segment of the carry-in job, or (ii) the window ends at the end of a segment of the carry-out job.

Formally, we consider the worst-case workload of a task τ_i in a window of length L, taking the maximum workload of the considered task, over all possible configurations in which the window is shifted right from the original configuration by $a \in \Gamma_1 \cup \Gamma_2$, where Γ_1 and Γ_2 are the sets of significant offsets to check corresponding to scenario (i) and (ii), respectively (see Figure 6).

Before deriving the formal offset values to check, let $\eta_i(a, L)$ be the carry-out length for task τ_i in a window of length L and offset a. Then,

$$\eta_i(a, L) = \min\left(L, (L + R_i - P_i + a) \mod T_i\right).$$

We note that the meaningful offsets to consider in scenario (i) correspond to the best-case starting times of each segment $\sigma_{i,j}$ of τ_i, i.e., $\sum_{x=1}^{j} P_{i,x}, \forall j \in [1, s_i]$. Moreover, all offsets greater than $P_i - \eta_i(0, L)$ can be ignored, since they would cause the end of the window to fall beyond the end of the carry-out job, resulting in a smaller workload. Therefore,

$$\Gamma_1 \doteq \left\{ \sum_{x=1}^{j} P_{i,x} \leq P_i - \eta_i(0, L), \forall j \in [1, s_i] \right\}.$$

The offsets to consider in scenario (ii) correspond to the difference (when positive) between the best-case starting times of each segment $\sigma_{i,j}$ and the original carry-out length $\eta_i(0, L)$, i.e.,

$$\Gamma_2 \doteq \left\{ \max\left(0, \sum_{x=1}^{j} P_{i,x} - \eta_i(0, L)\right), \forall j \in [1, s_i] \right\}.$$

5.2 Decomposing the carry-out job

One last observation concerns *predictability*, as defined in [20][4]. A schedulability test needs to be predictable, in that it should consider all possible execution times of a task system, as long as they do not exceed the given worst-case execution time. In other words, we would like the response-time provided by our analysis to be sufficiently robust to consider all possible execution requirements of the given tasks, including when some segment $\sigma_{i,j}$ requires less than $C_{i,j}$ time-units, or when a task may skip some of the segments. A schedulability test that does not properly consider situations when execution requirements are reduced is by no means sufficiently robust for critical applications.

The problem with the above approach is that a larger workload may fit the considered window if the carry-out skips some segment. Consider the example in Figure 4. In the upper scenario, the original situation is depicted, with the carry-out job contributing to the workload in the window of interest with its first two segments. However, when the second segment of the carry-out job is skipped, a worse situation is found, as shown in the lower part of the figure, since a segment with a higher parallelism may enter the window, resulting in a larger workload.

Considering all possible combinations of execution times appears overly complicated as it requires a combinatorial

[4] In [21], a broader concept is defined, i.e., "sustainability", which generalizes the notion of predictability.

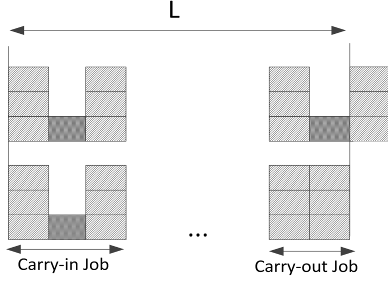


Figure 4: Densest possible packing of threads when a task skips some segment.

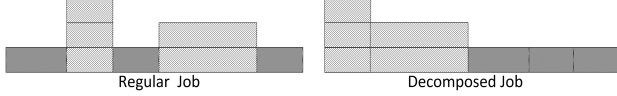


Figure 5: Example of a decomposed job

exploration of the possible segment instances of each task. To solve this problem, by allowing our analysis to be sufficiently robust, we will consider a pessimistic situation in which the carry-out job is decomposed, re-aligning the parallel segments such that the segments with higher parallelism are shifted to the beginning of the job's execution. Thus, segments are ordered by their number of p-jobs following a non-increasing pattern where segments with a higher number of p-jobs execute first, as depicted in Figure 5.

Replacing the original carry-out job by a decomposed job results in placing the parallel segments with higher parallelism within the window of interest, which allows us to obtain a sound upper bound on the workload of the carry-out job.

We are now ready to derive an upper bound of the workload that each task may impose on a window of length L.

5.3 Workload of a task within a window

Before presenting the analytical derivation of the workload components, we introduce the notion of "at least p-depth workload".

Definition 5. *The* at least p-depth workload *of a task τ_i in a window of length L, denoted as $W_i^p(L)$, is the sum of all intervals in which at least p threads of τ_i execute simultaneously in parallel.*

Note that the following relation holds by the definition of $I_{i,k}^p(L)$:

$$I_{i,k}^p(L) \le W_i^p(L).$$

The above relation, together with Theorem 2, gives the following lemma.

Lemma 4. *Given a set of synchronous parallel tasks τ scheduled by any work-conserving algorithm on m identical cores, the worst-case response-time of each task τ_k can be upper bounded by R_k^{ub} if*

$$\sum_{\forall \tau_i} \sum_{p=1}^{m} \min\left(W_i^p(R_k^{ub}), R_k^{ub} - P_k + 1\right) < m(R_k^{ub} - P_k + 1)$$

It now only remains to derive an upper bound on $W_i^p(L)$. We will compute such an upper bound by considering the at least p-depth contributions of carry-in, body and decomposed carry-out of each task τ_i in the worst-case scenario summarized in Figure 6, for all significant offsets $a \in \Gamma_1 \cup \Gamma_2$.

To compute the at least p-depth workload of the decomposed carry-out job, it is necessary to consider the first $\eta_i(a, L)$ units of the decomposed carry-out job. The following function computes the at least p-depth workload executed within the first x units of a generic job of τ_i.

$$g_i^p(x) = \begin{cases} 0, & \text{if } x \le 0 \\ \sum_{j=1:m_{i,j} \ge p}^{z} P_{i,j} + (x - \sum_{j=1}^{z} P_{i,j}), & \text{if } 0 < x \le P_i \text{ and } m_{i,z+1} \ge p \\ \sum_{j=1:m_{i,j} \ge p}^{z} P_{i,j}, & \text{if } 0 < x \le P_i \text{ and } m_{i,z+1} < p \\ \sum_{\forall j:m_{i,j} \ge p} P_{i,j}, & \text{otherwise,} \end{cases} \tag{8}$$

where z represents the index of the last segment that is fully included in the interval, so that $(z + 1)$ is the index of the segment that may execute partially within the carry-out interval.

The number of body jobs of τ_i executing in L is given by

$$\beta_i(L) = \left\lfloor \frac{L + R_i - P_i}{T_i} \right\rfloor - 1. \tag{9}$$

Note that $\beta_i(L)$ does not depend on a because the range in which a is varied never influences the number of body jobs. The at least p-depth workload of the body jobs of τ_i executing in L is then given by

$$b_i^p(L) = \beta_i(L) \sum_{\forall j:m_{i,j} \ge p} P_{i,j}. \tag{10}$$

The carry-in length $\alpha_i(a, L)$ can be derived as[5]

$$\alpha_i(a, L) = L - \eta_i(a, L) - \beta_i(L) T_i.$$

The at least p-depth carry-in contribution can then be derived by computing the workload executed within the last $\alpha_i(a, L)$ units of the carry-in job. The following function (from [7]) computes the at least p-depth workload executed within the last x units of a job of τ_i.

$$f_i^p(x) = \begin{cases} 0, & \text{if } x \le 0 \\ \sum_{j=h:m_{i,j} \ge p}^{s_i} P_{i,j} + (x - \sum_{j=h}^{s_i} P_{i,j}), & \text{if } 0 < x \le P_i \text{ and } m_{i,h-1} \ge p \\ \sum_{j=h:m_{i,j} \ge p}^{s_i} P_{i,j}, & \text{if } 0 < x \le P_i \text{ and } m_{i,h-1} < p \\ \sum_{\forall j:m_{i,j} \ge p} P_{i,j}, & \text{otherwise,} \end{cases} \tag{11}$$

where h represents the index of the earliest segment that is fully included in the interval, so that $(h - 1)$ is the index of

[5]When $R_i = P_i$ and $L \ge T_i$, the first job of τ_i executing in the window of interest is accounted for in the carry-in and not in the body contribution despite it has both release time and deadline within the window.

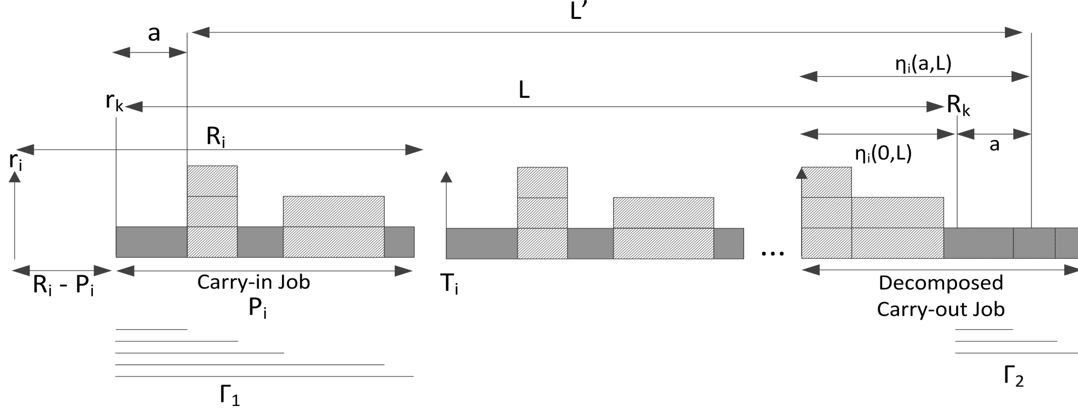


Figure 6: Response-time analysis details

the segment that may execute partially within the carry-in interval.

Considering Equation 8, Equation 10 and Equation 11, an upper bound on the at least p-workload of a task τ_i in a window of length L and offset a is given by:

$$\widehat{W}_i^p(L,a) = f_i^p(\alpha_i(a,L)) + b_i^p(L) + \tilde{g}_i^p(\eta_i(a,L)), \qquad (12)$$

where $\tilde{g}$ denotes that the function g is applied to the decomposed job. An upper bound on the worst-case workload of τ_i with depth at least p in a window of length L is then derived as

$$\widehat{W}_i^p(L) = \max_{a \in \Gamma_1 \cup \tilde{\Gamma}_2} \left\{ W_i^p(L,a) \right\}, \qquad (13)$$

where $\tilde{\Gamma}_2$ denotes that the offsets in this set are computed, again, considering the decomposed job.

Note that the above expression can be used to bound the inter-task workload from interfering tasks.

Before applying Lemma 4, a bound should also be provided to the intra-task interference, accounting for the workload of p-jobs from the same task. An upper bound on the intra-task workload of task τ_k with depth at least p can be given by:

$$\widehat{W}_k^p = \sum_{\forall j: m_{k,j} \geq p+1} P_{k,j}, \qquad (14)$$

where the sum is extended over all segments with parallelism at least $p+1$ instead of p since the p-jobs of the critical threads do not contribute to the critical interference.

5.4 Schedulability condition

Given the worst-case inter-task and intra-task workloads presented in the previous sections, we are now in a position for deriving an upper bound on the worst-case response time of a parallel task.

Lemma 5. *Given a set of synchronous parallel tasks τ scheduled by any work-conserving algorithm on m identical cores, the worst-case response-time of each task τ_k can be upper bounded by R_k^{ub} if*

$$\sum_{\forall \tau_{i \neq k}} \sum_{p=1}^{m_i} \min\left(\widehat{W}_i^p(R_k^{ub}), R_k^{ub} - P_k + 1\right) + \sum_{p=1}^{m_k} \min\left(\widehat{W}_k^p, R_k^{ub} - P_k + 1\right) < m(R_k^{ub} - P_k + 1).$$

Proof. The proof simply follows from Lemma 4, using the derived upper bounds instead of the real p-depth workload, and extending the p-indexed sum over the maximum number of p-jobs of each task[6]. □

For the special case of global fixed-priority scheduling, the interfering workload may be limited to the set of tasks having higher priority than τ_k. The following theorem can then be used to derive R_k^{ub} in a fixed priority setting.

Theorem 3. *Given a set of synchronous parallel tasks τ scheduled by global fixed-priority on m identical cores, an upper bound R_k^{ub} on the worst-case response-time of a task τ_k can be derived by the fixed-point iteration of the following expression, starting with $R_k^{ub} = P_k$:*

$$R_k^{ub} \leftarrow P_k + \left\lfloor \frac{1}{m} \left(\sum_{\forall i < k} \sum_{p=1}^{m_i} \min\left(\widehat{W}_i^p(R_k^{ub}), R_k^{ub} - P_k + 1\right) + \sum_{p=1}^{m_k} \min\left(\widehat{W}_k^p, R_k^{ub} - P_k + 1\right) \right) \right\rfloor.$$

Proof. If the iteration ends before R_k^{ub} reaches D_k, it is easy to see that the condition of Lemma 5 is satisfied, proving the theorem. □

A similar theorem holds for general work-conserving scheduling algorithms, extending the outer sum to all tasks $\tau_{i \neq k}$.

A schedulability test for systems scheduled with global fixed-priority is easily derived by computing R_k^{ub} for each task τ_k *in priority order*, starting from the highest priority

[6] As in [7], we are not taking advantage of the fact that carry-in and carry-out contributions may be less dense than in the considered scenario when there is some segment $\sigma_{i,j}$ with a parallelism $m_{i,j}$ greater than the number of processors m.

one, and checking whether $R_k^{ub} \leq D_k$ for all tasks. If not, the test is not able to guarantee the schedulability of the system. Note that, updating response time upper bounds in priority order allows optimally exploiting Theorem 3, since every task can use the most updated response times of the higher priority tasks, leading to smaller inter-task interferences.

5.5 Complexity

The complexity of the proposed response-time analysis is pseudo-polynomial in the task parameters, as is the original response-time analysis for sequential task sets presented in [8]. However, with respect to the sequential analysis, an additional s_i term has to be considered to account for the sliding window technique that repeats the workload computation for all segment starting times of the carry-in and carry-out jobs.

To obtain a faster analysis, a simple method is to consider the complete execution of the carry-in and carry-out job instances. To do that, it is sufficient to replace $\widehat{W}_i^p(L)$ in Theorem 3 with the following term:

$$\widehat{W}_i^p(L) = \left(\left\lfloor \frac{L + R_i - P_i}{T_i} \right\rfloor + 1 \right) \sum_{\forall j : m_{i,j} \geq p} P_{i,j}. \qquad (15)$$

As we will show in the experimental section, this method allows obtaining a faster worst-case response time computation without significant schedulability losses.

6. EVALUATION

This section presents the simulation results to evaluate the behaviour of our schedulability analysis, comparing it to other approaches existing in the literature. We only show the results for the implicit deadline case, which are however representative of the general behaviour. Concerning the simulation environment, we use a similar setting as in [7]. We start by generating a task set with m tasks, creating new task sets by adding a new task to the previous one until the task set utilization exceeds the number of processors. The above procedure is repeated until 40,000 task sets are generated.

The percentage of parallel tasks in the task set is controlled by a parameter that generates a random number from 0% to 100%. Periods of sequential tasks are uniformly generated in $[100, 1000]$, with C_i uniformly chosen from $[1, T_i]$. For parallel tasks, the number of segments s_i is uniformly generated in $[1, 5]$; the number of threads per segment $m_{i,j}$ is uniformly generated in the interval $[1, 3m/2]$; the worst-case execution times of the threads in each of the segments is uniformly chosen in the interval $[1, T_i/s_i]$; periods are uniformly generated in $[100, 10000]$.

For the generated task sets, we compare the number of schedulable task sets detected by our analysis (PAR-RTA) with the approach proposed in [7], denoted as PAR-EDF. In the same paper, the authors compare their test with other existing approaches that use a decomposition technique to schedule parallel tasks, and show that PAR-EDF outperforms all of them. We also show the performance of the faster method (PAR-RTA-UP) presented in Section 5.5 that uses the workload upper bound in Equation (15).

Figure 7a shows the results for $m = 4$. Both our approaches clearly outperform PAR-EDF, detecting 230% more schedulable task sets. Interestingly, the faster method using the simplified upper bound has a performance very similar to the complete method (within 1%)[7]. Increasing the number of processors, the situation is similar. Figure 7b shows the case with $m = 8$. While the number of schedulable task sets detected by all tests decreases, the relative performances remain the same.

7. CONCLUSION

Parallel task models are becoming important in the real-time systems community due to the recent shift to multi and many-core architectures, as well as the increase in the ubiquity of parallel programming models and frameworks. This paper contributes by filling the schedulability gap of synchronous parallel tasks by presenting an improved schedulability analysis for globally scheduled fixed-priority systems.

More specifically, a response-time analysis has been proposed for work-conserving schedulers, detecting the worst-case scenarios leading to the largest possible interference. A first test based on a sliding window technique and carry-out decomposition has been proposed. Then, a simplified test has been presented with a smaller computational complexity and comparable performance. Both tests are shown to significantly improve over the state of the art, in terms of number of schedulable task sets detected among randomly generated workloads.

Different future works are foreseen. We believe that the analysis could be refined by reducing the number of carry-in instances to consider, exploiting techniques presented in [22] for the sequential task model. Also, we intend to extend the response-time analysis framework presented in this paper to other task models (DAG-based, arbitrary deadlines, etc.) and scheduling policies, including global EDF.

8. ACKNOWLEDGMENTS

This research work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within projects Ref. FCOMP-01-0124-FEDER-037281 (CISTER), ref. FCOMP-01-0124-FEDER-020447 (REGAIN); by the European Union, under the Seventh Framework Programme (FP7/2007-2013), grant agreement n° 611016 (P-SOCRATES); also by FCT and by ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH / BD / 88834 / 2012.

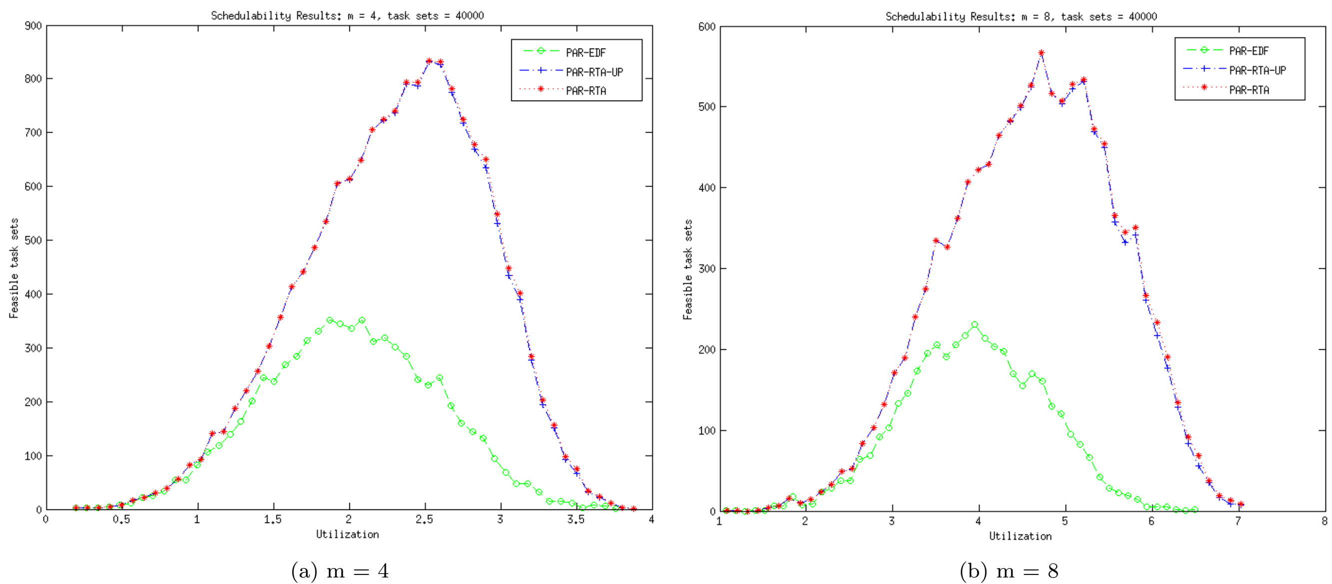
9. REFERENCES

[1] K. Asanovic *et al.*, “The landscape of parallel computing research: A view from berkeley,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[2] R. I. Davis and A. Burns, “A survey of hard real-time scheduling for multiprocessor systems,” *ACM Comput. Surv.*, vol. 43, no. 4, pp. 35:1–35:44, Oct. 2011.

[3] K. Lakshmanan, S. Kato, and R. R. Rajkumar, “Scheduling parallel real-time tasks on multi-core

[7] We found a similar result for sequential task sets, comparing the test in [8] with a pessimistic version that accounts for a complete carry-out contribution.



(a) m = 4

(b) m = 8

Figure 7: Number of schedulable task sets detected by the considered tests.

processors," in *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, ser. RTSS '10, 2010, pp. 259–268.

[4] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," *Real-Time Systems Symposium, IEEE International*, vol. 0, pp. 217–226, 2011.

[5] D. Lea, "A java fork/join framework," in *Proceedings of the ACM 2000 conference on Java Grande*, ser. JAVA '00, 2000, pp. 36–43.

[6] OpenMP, "Openmp," http://openmp.org/, Apr. 2014.

[7] H. S. Chwa, J. Lee, K.-M. Phan, A. Easwaran, and I. Shin, "Global EDF schedulability analysis for synchronous parallel tasks on multicore platforms," in *ECRTS*, 2013, pp. 25–34.

[8] M. Bertogna and M. Cirinei, "Response-time analysis for globally scheduled symmetric multiprocessor platforms," in *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, 2007, pp. 149–160.

[9] P. Axer *et al.*, "Response-time analysis of parallel fork-join workloads with real-time constraints," in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, July 2013, pp. 215–224.

[10] J. Goossens and V. Berten, "Gang FTP scheduling of periodic and parallel rigid real-time tasks," *CoRR*, vol. abs/1006.2617, 2010.

[11] C.-C. Han and K.-J. Lin, "Scheduling parallelizable jobs on multiprocessors," in *IEEE Real-Time Systems Symposium*, 1989, pp. 59–67.

[12] M. Drozdowski, "Real-time scheduling of linear speedup parallel tasks," *Inf. Process. Lett.*, vol. 57, no. 1, pp. 35–40, Jan. 1996.

[13] G. Manimaran, C. S. R. Murthy, and K. Ramamritham, "A new approach for scheduling of parallelizable tasks in real-time multiprocessor systems," *Real-Time Syst.*, vol. 15, no. 1, pp. 39–60, Jul. 1998.

[14] S. Kato and Y. Ishikawa, "Gang EDF scheduling of parallel task systems," in *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, ser. RTSS '09, 2009, pp. 459–468.

[15] K. Jansen, "Scheduling malleable parallel tasks: An asymptotic fully polynomial-time approximation scheme," in *Proceedings of the 10th Annual European Symposium on Algorithms*, ser. ESA '02, 2002, pp. 562–573.

[16] S. Collette, L. Cucu, and J. Goossens, "Integrating job parallelism in real-time scheduling theory," *Inf. Process. Lett.*, vol. 106, no. 5, pp. 180–187, May 2008.

[17] M. Korsgaard and S. Hendseth, "Schedulability analysis of malleable tasks with arbitrary parallel structure," *Real-Time Computing Systems and Applications, International Workshop on*, vol. 1, pp. 3–14, 2011.

[18] Q. Wang, , and G. Parmer, "Fjos: Practical, predictable, and efficient system support for fork/join parallelism," in *Proceedings of the 2014 IEEE 20th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[19] W. A. Horn, "Some simple scheduling algorithms," *Naval Research Logistics Quarterly*, vol. 21, no. 1, pp. 177–185, 1974.

[20] R. Ha and J. Liu, "Validating timing constraints in multiprocessor and distributed real-time systems," in *Proceedings of 14th IEEE International Conf. Distributed Computing Systems*, June 1994, pp. 162–171.

[21] S. Baruah and A. Burns, "Sustainable scheduling analysis," in *Proceedings of the IEEE Real-time Systems Symposium*, December 2006.

[22] N. Guan, M. Stigge, W. Yi, and G. Yu, "New response time bounds for fixed priority multiprocessor scheduling," in *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, 2009, pp. 387–397.