Overhead-aware schedulability evaluation of semi-partitioned real-time schedulers

Pedro Souto*‡, Paulo Baltarejo Sousa*†, Robert I. Davis§§, Konstantinos Bletsas*†, and Eduardo Tovar*†
*CISTER/INESC-TEC Research Center † ISEP-Polytechnic Institute of Porto, Portugal
‡ University of Porto, FEUP-Faculty of Engineering, Portugal § University of York, UK
Email: †{pbs, ksbs,emt}@isep.ipp.pt, ‡pfs@fe.up.pt §rob.davis@york.ac.uk

***Abstract*—Schedulability analyses, while valuable in theoretical research, cannot be used in practice to reason about the timing behaviour of a real-time system without including the overheads induced by the implementation of the scheduling algorithm. In this paper, we provide an overhead-aware schedulability analysis based on demand bound functions for two hard real-time semi-partitioned scheduling algorithms, EDF-WM and C=D. This analysis is based on a novel implementation that uses a global clock to reduce the overheads incurred due to the release jitter of migrating subtasks. The analysis is used to guide the respective off-line task assignment and splitting procedures. Finally, results of an evaluation are provided highlighting how the different algorithms perform with and without a consideration of overheads.**

I. Introduction

In multiprocessor real-time scheduling, the potential for tasks to migrate from one processor to another adds an extra dimension to the scheduling problem. Scheduling algorithms range from global algorithms, which allow unrestricted migration, to partitioned algorithms, which permit no migration at all [19]. Several approaches have been proposed that lie between these two extremes. This paper investigates two such algorithms, EDF-WM [24] and C=D [15]. In particular, we consider their implementation and detailed schedulability analysis, including a consideration of the overheads they induce.

When scheduling algorithms are first proposed and a schedulability analysis given, overheads are often disregarded or assumed to be negligible. In practice, however, when assessing the schedulability of a real-time system it is crucial that the overheads induced by the implementation of the algorithm are correctly accounted for. Without accurate accounting for such overheads schedulability tests may give false positives, indicating that a system is schedulable when in fact it is not, with obvious undesirable consequences.

Alternatively, it is often said that *overheads are assumed to be subsumed into the worst-case execution times (WCET) of tasks*. While this can result in valid overhead-oblivious analyses, there are two potential problems with such an approach. Firstly, certain overheads may result in additional release jitter, blocking or non-preemptive regions which need to be accounted for in other ways than just an inflated WCET, if the test is to remain valid. Secondly, inclusion of overheads within the WCET of tasks cannot accurately capture the real-time behaviour of a system that includes interrupts used to release tasks (or sub-tasks) on different processors.

This work was partially supported by FCT/MEC (Portuguese Foundation for Science and Technology) and when applicable, by ERDF (European Regional Development Fund) under the PT2020 Partnership within project UID/CEC/04234/2013 (CISTER Research Centre); by FCT/MEC and ERDF through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within project FCOMP-01-0124-FEDER-020447 (REGAIN); by FCT/MEC and the EU ARTEMIS JU within project ARTEMIS/0001/2013 - JU grant 621429 (EMC2). The research described in this paper was funded in part by the UK ESPRC grant, MCC (EP/K011626/1). EPSRC Research Data Management: No new primary data was created during this study.

A. Related Work

The papers introducing both EDF-WM [24] and C=D [15] algorithms include overhead-oblivious schedulability evaluations based on computation of the *demand bound function (dbf)*, a technique first proposed by Baruah et al. [7], [6]. The evaluation of C=D uses the Quick convergence Processor-demand Analysis (QPA) [36].

Early overhead-aware schedulability analyses targeted fixed-priority uniprocessor algorithms: Rajkumar et al. [29] considered cycle-stealing caused by DMA, Katcher et al. [23] considered the operating system overheads, and Audsley et al. [3] considered the effects of jitter and blocking. The effect of interrupts on dynamic priority uniprocessor algorithms were addressed by Jeffay and Stone [22]. Spuri [34] extended the results on the effects of jitter and blocking [3] to dynamic priority algorithms and also took into account system overheads.

The cache line evictions caused by task preemption or migration may cause significant overheads referred to as cache related preemption delays (CRPD) or cache related preemption and migration delays (CPMD). In the fixed-priority uniprocessor scheduling domain, Busquets-Mataix et al. [16] extended [3] to incorporate these overheads. More recently, improved methods were introduced by Altmeyer et al. [1] and extended to EDF by Lunniss et al. [28].

Extensive overhead-aware evaluations of multiprocessing scheduling algorithms were carried out at UNC, Chapel Hill, following the methodology initially developed by Calandrino et al. [17], namely 1) implementing the algorithms under analysis on the LITMUS^{RT} platform, 2) experimentally measuring the overheads incurred by these implementations and 3) randomly generating numerous task sets, whose schedulability is checked using schedulability tests for each tested algorithm, modified to account for the overheads

measured. Calandrino et al. [17] evaluated global EDF (G-EDF) and partitioned EDF (P-EDF) and various other global scheduling algorithms implemented in LITMUS^{RT} on a 4-processor symmetric multi-processor (SMP). Brandenburg et al. [13] used a larger platform (32 logical CPUs) to test the scalability of the algorithms evaluated in [17]. Brandenburg and Anderson [12] evaluated 7 different implementations of G-EDF. Finally, Bastoni et al. [9] compared the schedulability of clustered EDF (C-EDF) with both G-EDF and P-EDF, and later examined semi-partitioned scheduling algorithms with a focus on CPMD [10].

Overhead-aware schedulability analysis and task assignment for slot-based semi-partitioned algorithms were provided by Sousa et al. [31], [33], [32]. The methodology used was similar to that of [17], except that the algorithms were implemented directly on Linux and overhead-aware dbf-based schedulability tests were specifically developed for the algorithms. Brandenburg et al. [14] also evaluated different methods of accounting for interrupts in the schedulability analysis of global scheduling algorithms using tests specifically developed for that purpose.

Bastoni et al. [10] evaluated the overhead-aware schedulability of EDF-WM; however, that work does not detail how the original overhead-oblivious scheduling tests [24] were modified to take into account the overheads, nor the adaptations required to the task splitting algorithm[1].

Another evaluation approach consists of executing randomly generated task sets on a multiprocessor platform running the scheduling algorithms under study. Lelli et al. [25] used this approach to compare G-EDF, P-EDF and C-EDF and also variants of rate monotonic scheduling, focusing on soft real-time systems. Dellinger et al. [20] performed similar experimental evaluations on a 48-core AMD platform with a different Linux version, focusing on the scalability of G-EDF, P-EDF and C-EDF.

Complementary work by Andersson et al. [2] and Wu et al. [35] focused on the overheads incurred on the execution time of tasks caused by the sharing of hardware resources such as caches, memory and the memory bus with tasks executing in other cores/processors.

B. Contribution and Organization

In this paper we provide detailed overhead-aware analyses for the semi-partitioned algorithms EDF-WM and C=D. These analyses account for the overheads to do with task and sub-task releases via interrupts and inter-processor interrupts, including their blocking and release jitter effects. We account for scheduling and migration overheads, the overheads of budget enforcement timer interrupts, and show how cache related preemption and migration delays can also be included. We note that some of these overheads could be included in task WCETs, which we make clear by including them in revised computation demands for each sub-task. Since different types of task and sub-task incur different overheads, such an approach results in greater precision.

In Section II we present relevant background information and outline an implementation of both EDF-WM and C=D based on a novel approach that uses a global clock to reduce the jitter and overheads incurred due to migrating subtasks. In Sections III and IV we develop a detailed overhead-aware dbf-based schedulability analysis for EDF-WM and for C=D, respectively. A detailed experimental evaluation of these overhead-aware schedulability tests is provided in Section V, which shows how the relative performance of the algorithms changes when overheads are taken into account. Section VI concludes with a summary and discussion of future work.

[1]Brandenburg's thesis [11] offers some additional insights on the general approach used.

II. Background

A. System Model

We consider m identical processors and a set Γ of n independent sporadic tasks, $\tau_1,\ldots,\tau_n$. Each task τ_i generates a potentially unbounded number of *jobs* and is characterized by four parameters: the worst case execution time C_i, relative deadline D_i (with $0 \leq C_i \leq D_i$), and maximum release jitter J_i of each job; and the minimum inter-arrival time between consecutive jobs T_i.

B. Overhead-oblivious EDF Schedulability Analysis

Exact schedulability analysis for sporadic tasks scheduled on a uniprocessor using EDF relies on the use of the demand bound function proposed by Baruah et al. [7], [6]:

$$dbf(t) = \sum_{i=1}^{n} dbf(\tau_i, t) = \sum_{i=1}^{n} max\left(0, 1 + \left\lfloor \frac{t - Di}{T_i} \right\rfloor\right) \cdot C_i$$

A taskset is schedulable under EDF if and only if $\forall t, dbf(t) \leq t$.

The above analysis was extended by Spuri [34] to include tasks with release jitter and resource sharing under the Stack Resource Protocol (SRP) [4]. Zhang and Burns [37] consolidate these results providing the following schedulability test:

$$dbf(t) = b(t) + \sum_{i=1}^{n} n_i(t) \cdot C_i \leq t, \forall t \tag{1}$$

where $b(t)$ is the maximum time a job of a task τ_k with relative deadline $D_k \leq t$, can be blocked by a task τ_b whose deadline is $D_b > t$, and $n_i(t)$ is the number of jobs of task τ_i with release times and deadlines in an interval of length t:

$$n_i(t) = max\left(0, 1 + \left\lfloor \frac{t + J_i - D_i}{T_i} \right\rfloor\right)$$

where J_i is the release jitter of task τ_i.

Given that the demand bound function $dbf(t)$ changes its value only at discrete points, it is enough to check the inequality at those points. Furthermore, it has been shown that this inequality needs to be checked for an upper bounded interval of length L, equal to the longest busy period. Nevertheless, the number of points in which the

$dbf(t)$ changes its value in that interval can be very large. To speed up this analysis, Zhang and Burns [36] proposed the Quick convergence Processor-demand Analysis (QPA).

C. Overview of EDF-scheduler Implementation

We now outline an implementation of an EDF-scheduler on a uniprocessor. We assume this implementation later in the derivation of an overhead-aware schedulability analysis for EDF.

The EDF scheduler keeps a queue of jobs ready to execute, the *ready queue*, sorted by ascending absolute deadline. The scheduler picks the job at the head of the queue (i.e., the ready job with the earliest absolute deadline) to run.

When a job is released (by a timer interrupt, if it is periodic, or else by another interrupt), it is inserted into the ready queue. (This is akin to the approach suggested in [26].) When the interrupt returns, if the absolute deadline of the new job, i.e the one just released, is earlier than that of the interrupted job (if any), the scheduler is invoked and the new job is selected to execute next. The kernel then performs a context switch, the job which was previously running is preempted and the new job is started. Otherwise, the job that was previously running resumes its execution.

When a job terminates, it is removed from the ready queue and the scheduler is invoked. The scheduler then picks the job that is at the head of the ready queue, and the kernel performs a context switch starting the execution of that job.

D. Overhead-aware EDF Schedulability Analysis

In this section, we identify the run-time overheads induced by the "baseline" EDF scheduler and then integrate them into its schedulability analysis such that the analysis is valid by construction. (Note, here we do not consider cross-core interference or overheads that may arise from the sharing of hardware resources such as caches, main memory or the memory bus among processors [2], [35]).

From the implementation overview of EDF on a uniprocessor we can identify the following overheads and delays:

Release overhead ($RelO$) is the processing required upon release of a job, namely the handling of the interrupt that releases the job and the insertion of the job into the ready queue. We assume that interrupts are disabled during $RelO$, which we model as an immediate processing demand.

Scheduler overhead ($SchedO$) is incurred whenever the scheduler runs. Because in the implementation outlined above there may be a context switch every time the scheduler runs, this overhead also includes saving the context of the completed or preempted job and restoring the context of the job chosen to run next, including memory management unit registers, and, possibly, the invalidation of the translation lookaside buffer (TLB). We assume that interrupts and hence preemptions are disabled while the scheduler runs. With EDF scheduling this overhead occurs at most twice per job: when the job is released and when it terminates. Hence, we account for this overhead ($2 \cdot SchedO$) by adding it to the computation demand of each job and also including its blocking effect.

Cache related preemption delay (CRPD) is caused when one job preempts another. This may lead to eviction of the preempted job's cache lines, which will have to be fetched again when that job is resumed. This leads to a longer execution time than the WCET, C_i, because this parameter is typically obtained assuming no preemption [27]. Because, in the worst case, the preempted job may be resumed as soon as the preempting job terminates, we simply account CRPD as increasing the computation of the preempting job (C_i). This overhead depends in a complex way on the preempted and preempting tasks [28]. Tight estimation is out of the scope of this paper, instead, we assume an upper bound, $CprdO_i$, for the CRPD that each job τ_i may inflict on all other jobs it preempts (directly or indirectly).

Interrupt/preemption blocking (IpB), this is a delay that the release of a task may suffer, with respect to its arrival (indicated by the appropriate interrupt being raised). This delay is caused by the disabling of interrupts or the disabling of preemption during the normal execution of the currently running task.

Fig. 1 illustrates these overheads and how they are taken into account in our model. Initially, task τ_1 is running. At time t_1, task τ_2 is released with some blocking delay after its arrival time. The blocking delay is due to interrupts being disabled by task τ_1 for IpB and is shown as a crossed box on the timeline for τ_2. Because its absolute deadline is earlier than that of task τ_1, the scheduler runs and decides to preempt τ_1. After a context switch, τ_2 runs until completion. Again the scheduler runs and decides to resume task τ_1. This task starts to run at time t_2, after the context switch. Because τ_1 was preempted, some (or all) of its cache lines may have been evicted, and therefore it will incur a CRPD. As mentioned above, we charge this overhead to the preempting task and therefore this overhead is shown as a dashed rectangle in the timeline corresponding to the execution of τ_1. The CPRD may be spread throughout the full execution of task τ_1, but we assume the worst case (i.e. that it occurs immediately). Later, at time t_3, task τ_3 is released and inserted in the ready queue. Since its absolute deadline occurs after that of τ_1, the scheduler does not run, and task τ_1 is resumed after the interrupt. We assume that interrupt handling (IH) does not cause any CRPD, either because the cache is disabled or because interrupt handlers use a separate cache partition.

Note that in our analysis, we do not consider the overheads required to manage periodic scheduler-tick interrupts, since event-driven schedulers such as EDF do not make use of those interrupts [10].

We now amend the dbf given in (1) to take into account the overheads of the EDF implementation described above, as follows:

$$dbf(t) = b(t) + \sum_{i=1}^{n} n_i(t) \cdot C'_i + \sum_{i=1}^{n} RelI_i(t) \qquad (2)$$

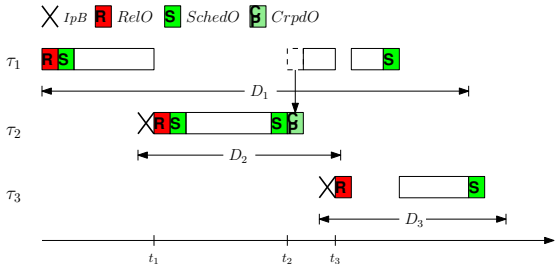

Figure 1: Time diagram of overheads in uniprocessor EDF.

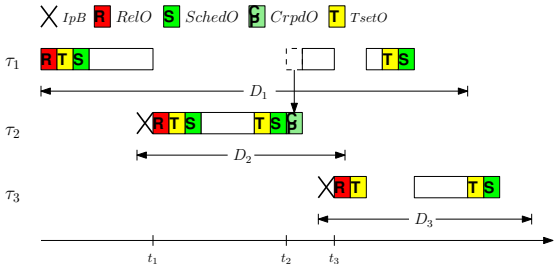

Figure 2: Time diagram of overheads in uniprocessor EDF with budget enforcement timers.

where

$$b(t) = \begin{cases} max(IpB, SchedO) & \text{if } t < max_{i=1}^{n} D_i \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

$$C_i' = C_i + 2 \cdot SchedO + CprdO_i \quad (4)$$

$$RelI_i(t) = \left\lceil \frac{t + J_i}{T_i} \right\rceil \cdot RelO \quad (5)$$

Here, $b(t)$ is the maximum blocking due to disabling interrupts during either normal task execution, i.e. IpB, or scheduler operation, i.e. $SchedO$. We assume that every task effectively includes scheduler operations and can cause the same maximum blocking due to disabling interrupts, since such critical sections typically only occur during system calls. For simplicity, we assume that there are no software resources shared among tasks. The blocking effects caused by accessing such resources under SRP can however easily be incorporated into our model via their inclusion in the blocking factor $b(t)$.

Note that (4) is safe but may be very pessimistic, as it is tantamount to assuming that every job release leads to a preemption. A tighter demand bound can be achieved by using a tighter bound on the number of preemptions [30][21].

Since all of the overheads and delays are included within the blocking and execution time terms, the worst-case scenario for baseline EDF also applies to the augmented equations. Note that the interference $RelI_i$ caused by the release of a job via the respective interrupt handler is accounted for by independently maximizing the interference from it in any given time interval t in (5), thus ensuring that the overall value for $dbf(t)$ remains a valid upper bound.

E. Budget Enforcement Timers

In order to ensure that the processor utilization by a task does not exceed its upper bound, it is common to use a budget enforcement timer, or *budget timer* for short, per task. Such a timer can be implemented with the help of standard kernel operations for setting and cancelling timers, by using a function to read the current time and a variable per task used to keep its remaining execution time. Furthermore, for quick handling of their expiry, the timers can be maintained in a queue sorted by ascending expiry time. This is similar to the timer described in [24] for the implementation of timed (sub)tasks in EDF-WM.

The handling of the budget timer leads to one additional overhead, the *timer setup overhead* ($TsetO$). This comprises both the overhead required to cancel the timer of the previously executing task and to setup the budget timer of the next task. Although these overheads do not occur one after the other, we lump them together (see Fig. 2) for ease of representation, and also to simplify the expressions used in the analysis.

The timer setup overhead has two main effects. First, it increases the interval during which interrupts may be disabled by $TsetO$. Therefore, we amend the blocking term, $b(t)$, given in (3) to:

$$b(t) = \begin{cases} max(IpB, SchedO + TsetO) & \text{if } t < max_{i=1}^{n} D_i \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

Second, as shown in Fig. 2, each job may incur this overhead twice: once upon its release and again upon its termination. Therefore we amend (4) and (5), respectively, as follows:

$$C_i' = C_i + 2 \cdot SchedO + TsetO + CprdO_i \quad (7)$$

$$RelI_i(t) = \left\lceil \frac{t + J_i}{T_i} \right\rceil \cdot (RelO + TsetO) \quad (8)$$

An alternative to a budget timer is a deadline-enforcement timer. While the former ensures that a job does not exceed its allocated processing time, the latter ensures that a job does not execute past its deadline. Deadline-enforcement timers are simpler to implement, but it is not possible to implement timed (sub)tasks in EDF-WM (see below) with them, hence the use of budget timers here.

For ease of reference, Table I lists all the symbols defined in this and the following sections and provides a short description of their meaning. A more detailed explanation is provided at the appropriate point in the text.

In the next two sections we extend the above analysis for both EDF-WM and C=D. First, we give an overview of the implementation of both algorithms.

F. EDF-WM and C=D

Both the EDF-WM and C=D algorithms use strict EDF on each processor. In both cases, tasks are partitioned by default. Only when this is not possible is a task "split",

Symbol	Meaning
τ_i	ith task in a task set
Γ	Task set
C_i	Worst-case execution time of τ_i
C'_i	Worst-case execution time including overheads of τ_i (except overheads that occur when τ_i is released)
D_i	Relative deadline of τ_i
J_i	Maximum release jitter of (periodic) task τ_i
p	A processor
T_i	Minimum inter-arrival time of task τ_i
$b(t)$	Maximum blocking of any job with relative deadline smaller or equal to t, by jobs whose relative deadline is larger than t
$dbf(t)$	Demand bound function for all tasks assigned to a processor in a time interval of duration t
$n_i(t)$	Maximum number of executions of τ_i in a time interval t
$RelI(t)$	Demand caused by the release of all tasks on a processor in a time interval t
$RelI_i(t)$	Demand caused by the release of τ_i in a time interval t
$IpiI(t)$	Demand caused by IPI used to set the release timer for all middle and last subtasks on a processor in a time interval t
π	Global clock precision
$BetO$	Upper-bound of the budget enforcement timer IH overhead
$CrpdO_i$	Upper-bound of the CRPD caused by τ_i
$CrmdO_s^j$	Upper-bound of the cache related migration delay of τ_s^j
$IntB_p$	Upper-bound of the blocking of an interrupt on p
$IntC_p$	Upper-bound of the delay of an interrupt on p because of interrupt contention
$IpiJ$	Upper-bound of the interprocessor interrupt (IPI) jitter
$IpiO$	Upper-bound of the IPI overhead
$MigrO$	Upper-bound of the migration overhead
IpB	Upper-bound of the blocking caused by disabling interrupts or task preemptions during normal execution of a task
$RelO$	Upper-bound of the release overhead
$RihR_s$	Maximum response time of the release IH of τ_s
$SchedO$	Upper-bound of the scheduling overhead (includes context switch overhead)
$TsetO$	Upper-bound of the timer setup overhead

Table I: Notation

Some parameters in this table appear also in the form par_s^j, to denote the parameter par of subtask τ_s^j of task τ_s, where j can be one of 1, m or ℓ, denoting respectively the first, a middle or the last subtask.

i.e. configured to migrate between selected processors. The execution of a task on different processors is conveniently described as a sequence of *subtasks* with precedence constraints. On its processor, a subtask is handled by the local EDF scheduler just like any other task. Where EDF-WM and C=D differ is in the algorithms used 1) to assign tasks to processors and chose tasks to split, and 2) to compute the scheduling parameters (C and D) of the subtasks of migratory tasks.

EDF-WM assigns tasks in order of non-increasing D_i. It first tries to assign a task as non-migrating using first-fit; if this fails, it then tries to assign the task by splitting. The number of subtasks is determined tentatively, starting with 2 and incrementing until success; if a task cannot be scheduled even if split into m subtasks (the number of processors), the task set is deemed unschedulable. EDF-WM assigns to all s subtasks of a given migratory task with deadline D the same deadline D/s. The execution time C_s^i of subtask i of migratory task τ_s, is the maximum computation that the respective processor can provide and still ensure the schedulability of that subtask and all other tasks already assigned to that processor. To avoid splitting tasks into too many subtasks, EDF-WM calculates the maximum amount of computation time that each processor can provide to the migratory task and then selects processors for the subtasks in decreasing order of that quantity.

Under **C=D**, on the other hand, all subtasks but the last of a migratory task have their relative deadlines equal to their computation time, hence the name "C=D". As a result, all subtasks, except possibly the last, run at the highest (task) priority on their respective processors. Several strategies and task orderings were explored for task assignment and splitting by the original authors [15]. Assuming no overheads, the best results were obtained by pre-selecting the tasks to split in non-decreasing D_i order and by assigning non-migratory tasks to processors in non-increasing density order. Like EDF-WM, the number of split tasks is determined tentatively. Unlike EDF-WM, when assigning subtasks, processors are selected by increasing index order.

G. EDF-WM/C=D Scheduler implementation

A characteristic of both EDF-WM and C=D is that all subtasks, but the last, of a migratory task are *timed*. That is, their termination occurs on expiry of their assigned (artificial) budget, rather than on completion of all of the task's computation. Timed subtasks can be easily implemented with budget timers. Indeed, as described in Sec. II-E, these timers are started when a subtask is scheduled to run for the first time, and are then suspended and resumed as the processor suspends and resumes execution of that subtask. Upon expiry of a subtask budget timer, the subtask is terminated. Thus, budget timers ensure that subtasks never overrun.

With budget timers, the implementation of an EDF-WM/C=D scheduler raises essentially one implementation issue that affects the schedulability analysis: the release of a subtask on another processor.

The release of a subtask on a remote processor can be implemented in a number of different ways. In our implementation, when a migratory task, i.e. its first subtask, is released, an interprocessor interrupt (IPI) is sent to each of the other processors that execute a subtask of that migratory task. The IPI handler on each of those processors then sets up a *release timer*. The release time set depends on which part of the migratory task the processor was assigned. Therefore, all subtasks of a migratory task, except possibly the first, are released by means of a timer interrupt. The release of a subtask upon expiry of the release timer is handled as per the release of any normal task under partitioned EDF.

To set up the release timer, we assume the availability of a high-resolution clock that can be read from any processor[2]. Before sending the IPI, the release handler of the first subtask reads this clock and puts the reading in a known memory location. The IPI handlers then read this value as well as the global clock, in order to determine the delay incurred by the IPI notification. This delay is then used to compute a corrected value with which to program the release timer for each subtask. We use a global clock (synchronised) approach in this way since it avoids the accumulation of release jitter along a chain of sub-tasks which would otherwise adversely affect schedulability.

By computing the appropriate release timer values, based on the analysis presented in the following sections, and by using upper bounds on the overheads, we can ensure that the subtasks precedence constraints are satisfied. Furthermore, any violation of these constraints can be easily detected e.g. by locking the task's data structure.

III. EDF-WM

In this section, we integrate the overheads introduced by the mechanisms used to implement EDF-WM into the overhead-aware analysis of uniprocessor EDF.

Overheads: From the implementation outlined in the previous section, we identify the following additional overheads, illustrated in Fig. 3.

Interrupt/preemption Blocking (IpB) Recall that this is the time for which interrupts may be disabled during normal task execution. In EDF-WM such blocking may delay handling of the budget timer interrupt, prolonging the execution time of a subtask and effectively increasing the job's execution demand. This effect can also occur in uniprocessor EDF with budget timers, when a job overruns its estimated WCET. However this case corresponds to a timing-fault, and hence was not taken into account in the overhead-aware uniprocessor analysis. With EDF-WM it needs to be accounted for since budget expiry is part of the normal operation of the system.

Budget (Enforcement) Timer Overhead ($BetO$) This is the overhead of handling a budget timer interrupt. It is incurred by all of the subtasks, except the last, of every migratory task. This adds to the job's demand. We assume that interrupts are disabled while the budget timer interrupt handler executes. Since the budget timer interrupt can only occur when the associated subtask is executing, we can include this overhead as if it were part of the subtask's execution time).

Migration overhead ($MigrO$) is the overhead required to move the task from the current processor to a "place" where the next processor can find it. This adds to the job's demand. In our implementation, this overhead occurs as part of the context switch. In Fig. 3 we represent it as occurring after the scheduling overheads. Note that interrupts are disabled during this time.

[2]E.g. recent x64 multicore processors, from either Intel or AMD, ensure that the Time Stamp Counter of the different cores are synchronized and increment at an invariant rate.

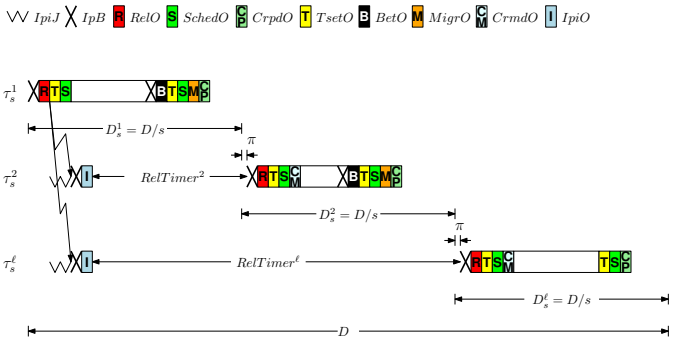

Figure 3: Time diagram of EDF-WM for an implementation based on release timers.

IPI jitter ($IpiJ$) is the jitter of the IPI delay, i.e. the maximum delay between the sending of an IPI on one processor and the raising of the IPI on the target processor. It contributes to the jitter of the *IPI overhead* (see below), but not to the release jitter of the subtask, since the implementation relies on a global clock that can be used to determine and compensate for the IPI delay.

IPI overhead ($IpiO$) is the time required to handle the IPI interrupt, including reading the global clock and setting up the release timer, and also suspending and later resuming the budget timer of the interrupted task. We assume that interrupts are disabled during this time interval. This IPI overhead is modeled in a way similar to the release overhead, i.e. immediately upon its occurrence, but taking into account its own jitter, rather than that of the task release.

Cache related migration delay (CRMD) is caused upon migration of a job, as it may have to fetch again its cache lines when it is resumed on another processor. As for the CRPD, computing a tight bound of this overhead is out of scope of this paper. Instead, we assume that there is a known upper bound for the CRMD of each migratory job, $CrmdO_s^j$.

In addition to these overheads, schedulability is also affected by the *precision* of the readings of the global clock, π. As the values of the release timers depend on these readings, their uncertainty π, contributes to some jitter in the release of all subtasks except for the first.

Processor Demand: Because different types of subtask (first, middle and last) incur different overheads, as shown in Fig. 3, we develop the processor demand for each subtask type. We consider first the effect of the overheads on the computation demand and then their effect on the jitter.

First subtask: This is a timed task, i.e. it terminates upon expiry of its budget timer and is followed by another subtask. Therefore the computation demand including overheads of the first subtask becomes:

$$C'^1_s = C^1_s + 2 \cdot SchedO + TsetO + CrpdO^1_s + IpB + BetO + MigrO \quad (9)$$

In comparison with partitioned EDF (see (7)), it includes $BetO$ and $MigrO$, as well as IpB. The latter is included since the subtask might have blocked interrupts/preemption and therefore overrun by up to IpB after the expiry of its budget timer.

Last subtask: This is like a non-migratory task, except that it incurs one additional $CrmdO_s^\ell$ because it has migrated from another processor and has to restore its cache lines. Therefore, the computation demand, including overheads, of the last subtask becomes:

$$C'^{\ell}_s = C_s^\ell + 2 \cdot SchedO + TsetO + CrpdO_s^\ell + CrmdO_s^\ell$$

Middle subtask: A middle subtask like the first subtask is timed and is followed by another subtask. It is also similar to the last subtask in that it "migrates" from another processor. Therefore, its computation demand, including overheads, becomes:

$$C'^{m}_s = C_s^m + 2 \cdot SchedO + TsetO + CrpdO_s^m + IpB + BetO + MigrO + CrmdO_s^m \quad (10)$$

Next we analyze the effect of the overheads on the release jitter. The release of either a middle or a last subtask suffers an additional jitter, comprising the *response time of the release interrupt handler* $RihR_s$, of the first subtask and the precision, π of the reading of the global clock. Therefore, J_i in (II-B) and (8) for all subtasks τ_s^i, that are not the first of the respective migratory task is replaced by:

$$J_s^i = J_s + RihR_s + \pi \quad (11)$$

$RihR_s$ comprises a blocking term $IntB_p$, caused by disabling interrupts and preemptions (note p is the processor of the first subtask), and another term $IntC_p$, caused by the contention among the different interrupt sources. $RihR_s$ is given by:

$$RihR_s = IntB_p + IntC_p \quad (12)$$

where $IntB_p$ is given by:

$$IntB_p = \begin{cases} max(IpB, SchedO + TsetO + MigrO), \text{ if there} \\ \quad \text{is another subtask in } p \text{ that is not the last} \\ max(IpB, SchedO + TsetO), \text{ otherwise} \end{cases}$$

and, $IntC_p$ is given by:

$$IntC_p = N_p \cdot max(RelO + TsetO, IpiO, BetO) \quad (13)$$

where N_p is the number of (sub)tasks in processor p.

There are three assumptions underlying (13). First, that no interrupt can reoccur within the short time interval $IntC_p$ (and hence fixed point iteration is not needed to compute $IntC_p$). Second, to ensure that $IntC_p$ is an upper bound, we assume that the release interrupt of interest is the lowest priority interrupt in its processor. Finally, we assume that the only interrupts are those used to implement task releases, budget enforcement and migration. (We note that other interrupts could easily be included by modifying the calculation of $IntC_p$).

Finally, the release of subtasks on remote processors induces one IPI overhead per release of middle or last subtasks on the respective processor. This is in addition to the release overhead, therefore we amend (2) as follows:

$$dbf(t) = b(t) + \sum_{i=1}^{n} RelI_i(t) + \sum_{i=1}^{n} n_i(t) \cdot C'_i + \sum_{\tau_s^i \in \Omega} IpiI_s^i(t) \quad (14)$$

where Ω is the set of either middle or last subtasks of migratory tasks, and the interference due to each of these tasks caused by IPI is:

$$IpiI_s^i(t) = \left\lceil \frac{t + J_s + RihR_s + IpiJ}{T_s} \right\rceil \cdot IpiO \quad (15)$$

and

$$b(t) = \begin{cases} 0, \text{ if there is no (sub)task, } \tau_i, \text{ s.t. } D_i \geq t \\ max(IpB, SchedO + TsetO + MigrO), \text{ if there is} \\ \quad \text{a subtask, } \tau_s^i, \text{ that is not the last s.t. } D_s^i \geq t \\ max(IpB, SchedO + TsetO), \text{ otherwise} \end{cases}$$

A more detailed analysis of the jitter incurred by migratory tasks, and the adaptation of the partitioning and splitting algorithm presented in the original EDF-WM paper [24] can be found in the appendices.

This analysis leads to a dbf-based test that is sustainable with respect to changes in the same task parameters as the overhead oblivious analysis [5]. Indeed, the early completion of a migratory task in a subtask different from the last, also leads to a lower demand on both the corresponding processor as well as on all processors that were assigned the following subtasks of that migratory task. It is also sustainable with respect to decreases in the size of the various overheads, since all of the terms that are added in the dbf are monotonically non-decreasing with respect to these values.

We note that due to the heuristics used in task allocation, it could occasionally be the case that a task set that is "easier" to schedule (smaller parameter values) is deemed unschedulable by the tests when one that is "harder" to schedule would be deemed schedulable due to a different processor allocation. Nevertheless, any task set that is deemed schedulable by the tests will be schedulable at run time even with smaller values for overheads, execution times etc.

IV. C=D Scheme

Because we assume identical implementations, the schedulability analysis developed for EDF-WM is also applicable to C=D.

However, to preserve the C=D notion, the deadline for the first subtask must be the earliest time by which its demand can be guaranteed to be supplied, even in the presence of delays caused by blocking, inter-processor interrupts and the release of other (sub)tasks assigned to the same processor. Thus, D_s^1 is given by:

$$D_s^1 = max(IpB, SchedO + TsetO) + C'^{1}_s + RelI(D_s^1) + IpiI(D_s^1) \quad (16)$$

where ${C'}_s^1$ is given by (9), $RelI(t)$ is an upper bound on the release interference in a time interval of duration t and is given by:

$$RelI(t) = \sum_{i:\tau_i \in \Gamma} RelI_i(t) \tag{17}$$

where $RelI_i(t)$ is given by (8), replacing J_i with the jitter expression for the appropriate type of subtask (i.e. from (11) for middle and last subtasks), and Γ is the set of tasks on the same processor. Note that this expression includes the release overheads of the C=D subtask itself. Finally, $IpiI(t)$ is an upper bound on the IPI interference in a time interval of duration t and is given by:

$$IpiI(t) = \sum_{i:\tau_i \in \Omega} IpiI_s^i(t) \tag{18}$$

where $IpiI_s^i(t)$ is given by (15) and Ω is the set of middle and last subtasks of migratory tasks on the same processor.

Computing $C's$ and $D's$ of subtasks: Although it would be possible to adapt the iterative algorithm described in Section 2.3 of [15] to compute the C's and the D's of the different subtasks, it is simpler and more intuitive to use binary search. Like in the original algorithm, we tentatively add the first subtask to the set of tasks already assigned to the processor under analysis. The algorithm is used to compute D_s^1, rather than C_s^1, because (16) is a fixed point iteration on D_s^1 and it is easier to derive C_s^1 from the latter than the other way around.

The initial interval is set to $[0, D_s]$, where D_s is the relative deadline of the task to split. In each iteration, we set D_s^1 to the midpoint of the current interval, compute the corresponding C_s^1, from (16) and (9), and run QPA to determine whether the taskset is schedulable, and adjust the interval as appropriate. When the width of the interval is 1, we assign the value of the lower end of the interval to D_s^1 and derive the corresponding C_s^1.

If C_s^1 is positive the splitting is successful and we set the parameters of the second subtask of the split (sub)task as:

$$C_s^2 = C - C_s^1$$
$$D_s^2 = D - D_s^1$$

Migratory tasks with more than two subtasks: In [15] the authors describe two strategies for selecting the tasks to split: the *continuous strategy* always leads to migratory tasks with only two subtasks, but the *pre-selection strategy* may lead to migratory tasks with more than two subtasks.

To account for middle subtasks, we need to make some adjustments. First, we compensate for the (lack of) precision of the global clock, by taking it into account in the computation of the deadline, i.e. by amending (16) of the middle subtask as follows:

$$D_s^m = \pi + max(IpB, SchedO + TsetO) + {C'}_s^m + RelI(D_s^m) + IpiI(D_s^m)$$

Second, C_s^m must be derived from (10) rather than (9).

V. Schedulability Experiments

In this section, we present an overhead-aware evaluation of the schedulability of different configurations of the EDF-WM and C=D algorithms. The main goal of this study is to see how the relative performance of these algorithms is affected when overheads are taken into account, and how this differs from the case where overheads are ignored. The metric used in this evaluation is the fraction of successfully scheduled task sets.

A. Random task set generation and overhead parameters

We considered a system with $m = 8$ processors and used the UUnifast-Discard [18] algorithm to generate task sets with utilisations in the range 5.6 to 7.9 (70% to 97.5% normalized utilisation) and uniformly distributed task utilisations. To investigate the effect of average task utilisation on algorithm performance, we explored three scenarios: $n = 12$, 16 or 24 (dubbed "heavy/medium/light tasks"). Task periods were uniformly distributed in the range [5, 50] ms, with a resolution of 1 ms. All tasks generated were implicit deadline ($D_i = T_i$). The WCET of each task was derived from its utilization and its period ($Ci = u_i \cdot T_i$).

The values used as upper bounds for the different overhead parameters are listed in Table II. These values, except for the cache-related delays and the clock reading precision, were determined experimentally on a preliminary implementation of the scheduler outlined in Sec. II-G on a 2.6.31 Linux kernel running on a platform of 24-cores built from two 1.9 GHz AMD Opteron 6168 processors, using an approach similar to that used in [10]. Specifically, we ran 100 randomly generated task sets (each with a randomly selected number of tasks) for 1000 seconds each. Given the unpredictability of our platform, we rounded up the worst case observed value for each overhead, after discarding outliers using an 1.5 inter-quartile range filter. The cache-related delays are taken from the values measured in [10]. The clock reading precision is a very safe estimate[3].

B. Results

Fig. 4 shows the results obtained for a system with $m = 8$ processors. Each point in these plots represents the schedulability success ratio for 500 task sets with the corresponding characteristics. The overall effectiveness of the different algorithms (with and without overheads) is also given by the *weighted schedulability* measures [8] in Table III. These values are for the same experiments and thus summarise the results illustrated in Fig. 4.

For EDF-WM, we ordered tasks by non-increasing deadline (EDF-WM(D)), as recommended by its authors, and by decreasing density (EDF-WM(DN)). For C=D, we tested both the "pre-selection" (C=D(Pre-sel)) and the "continuous" strategies (C=D(Cont)), both with decreasing density as the packing order and increasing deadline as the splitting

[3]Recent measurements on the same platform show that the clock reading precision is of the order of a few tens of ns.

Table II: Experimentally-derived values for the various scheduling overheads (in μs).

	π	$BetO$	$CrpdO$	$CrmdO$	IpB	$IpiJ$	$IpiO$	$MigrO$	$RelO$	$SchedO$	$TsetO$
Values	1	10	100	100	10	10	15	10	10	20	5

Table III: Weighted Schedulability.

n	Overheads	P-EDF(D)	P-EDF(DN)	EDF-WM(D)	EDF-WM(DN)	CD(Cont)	C=D(Pre-sel)
12	No	0.453	0.534	0.759	0.789	0.718	0.879
12	Yes	0.413	0.497	0.582	0.712	0.665	0.638
16	No	0.522	0.697	0.806	0.867	0.855	0.894
16	Yes	0.47	0.642	0.629	0.767	0.766	0.729
24	No	0.686	0.882	0.865	0.896	0.9	0.906
24	Yes	0.595	0.782	0.687	0.794	0.788	0.789

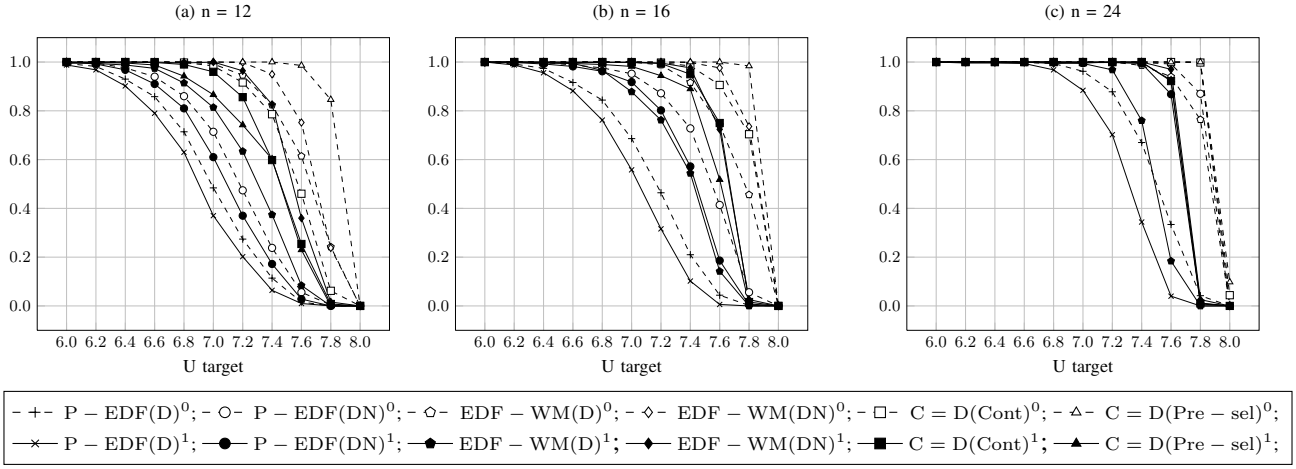

Figure 4: Schedulability ratio with and without overheads (superscripts 1 and 0, respectively) for $m = 8$.

order, see [15]. Partitioning (First-fit), is also provided for reference, both with decreasing density (P-EDF(DN)) and decreasing deadline (P-EDF(D)) used for task assignment.

We drawn the following conclusions from the results illustrated in Fig. 4. These results confirm some of the conclusions previously drawn from overhead-oblivious evaluation, while contradicting others.

(i) A higher average task utilisation (i.e. fewer, but "heavier" tasks) tends to make the task sets harder to schedule for all three semi-partitioned algorithms, as well as for the baseline partitioning approach. This is observed as an increase in schedulability as the number of tasks n increases (from 12 to 16 to 24). With many light tasks, it is the packing algorithm that is most important, with decreasing density being most effective, and there is little benefit to be obtained from splitting tasks. (The weighted schedulability (WS) is approximately equal for EDF-WM(DN), P-EDF(DN) and both C=D approaches at 0.79 for 24 tasks).

(ii) The semi-partitioned algorithms perform significantly better than pure partitioning, for task sets with medium or heavy tasks and crucially this remains the case when overheads are considered. In other words the additional overheads of the semi-partitioned approaches are more than made up for by improvements in schedulability. (The weighted schedulability of the semi-partitioned approaches is over 0.58 (with overheads), compared to 0.50 for P-EDF(DN)).

(iii) In general, for low n (a few heavy tasks) EDF-WM(DN) performs best. For medium n, the C=D(Cont) algorithm performs as well as EDF-WM(DN), and both perform better than C=D(Pre-Sel).

(iv) When overheads are considered, C=D(Cont) performs better than C=D(Pre-sel) with a weighted schedulability of 0.67 v. 0.64 for 12 tasks. This is in direct contrast to the overhead-oblivious case (see also Fig. 7 in [15]) where C=D(Pre-sel) outperforms C=D(Cont) with a weighted schedulability of 0.88 v. 0.72. The reason for this turnaround is that C=D(Cont) typically results in fewer split tasks and thus lower overheads than C=D(Pre-sel).

This final point highlights not only the importance of including overheads on the overall schedulability of different algorithms, but also how an appropriate consideration of overheads affects relative performance and thus the choice of which methods to deploy in real systems.

VI. Conclusions

The availability of overhead-aware schedulability analysis for multiprocessor scheduling algorithms is critical to reliably schedule hard real-time tasks using these algorithms.

In this paper we developed detailed overhead-aware schedulability analysis based on the demand bound function for two state-of-the-art semi-partitioned hard real-time multiprocessor scheduling algorithms, EDF-WM and C=D.

This analysis was founded on a detailed investigation of the implementation issues inherent in these two algorithms. Further, we showed how to apply this analysis to determine the tasks' scheduling parameters.

We used the overhead-aware schedulability analysis to perform an evaluation of the two algorithms, using measured values as upper bounds for the various overheads identified. We found that when overheads are accounted for, EDF-WM and C=D retain the significant advantages that they have over simple partitioning, particularly for systems with heavy (high utilisation) tasks. We evaluated two variants of C=D, one based on continuous selection of tasks to split and the other based on pre-selection. Our results showed that in direct contrast to earlier results ignoring overheads, the continuous selection approach was significantly more effective than the pre-selection approach when overheads are considered. We note that this is due to the former typically splitting fewer tasks and thus incurring lower overheads.

REFERENCES

[1] S. Altmeyer, R. Davis, and C. Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems*, 48(5):499–526, 2012.

[2] B. Andersson, A. Easwaran, and J. Lee. Finding an upper bound on the increase in execution time due to contention on the memory bus in cots-based multicore systems. *SIGBED Rev.*, 7(1), January 2010.

[3] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.

[4] T. Baker. Stack-based scheduling of realtime processors. *Real-Time Systems*, 3(1):67–99, March 1991.

[5] S. Baruah and A. Burns. Sustainable scheduling analysis. In *proceedings of the 27th Real-Time Systems Symposium (RTSS)*, 2006.

[6] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *proceedings of the 11th Real-Time Systems Symposium (RTSS)*, pages 182–190, 1990.

[7] S. Baruah, L. Rosier, and R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2:301–324, 1990.

[8] A. Bastoni, B. Brandenburg, and J. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, pages 33–44, Brussels, Belgium, 2010.

[9] A. Bastoni, B. Brandenburg, and J. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. In *proceedings of the 31st Real-Time Systems Symposium (RTSS)*, pages 14–24, 2010.

[10] A. Bastoni, B. Brandenburg, and J. Anderson. Is semi-partitioned scheduling practical? In *proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 125–135, 2011.

[11] B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, UNC at Chapel Hill, 2011.

[12] B. Brandenburg and J. Anderson. On the implementation of global real-time schedulers. In *proceedings of the 30th Real-Time Systems Symposium (RTSS)*, 2009.

[13] B. Brandenburg, J. Calandrino, and J. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *proceedings of the 29th Real-Time Systems Symposium (RTSS)*, 2008.

[14] B. Brandenburg, H. Leontyev, and J. Anderson. Accounting for interrupts in multiprocessor real-time systems. In *proceedings of the 15th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 273–283, 2009.

[15] A. Burns, R. Davis, P. Wang, and F. Zhang. Partitioned EDF scheduling for multiprocessors using a $C = D$ task splitting scheme. *Real-Time Systems*, 48(1):3–33, Jan 2012.

[16] J. Busquets-Mataix, J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *proceedings of the 2nd Real-Time and Applications Symposium (RTAS)*, 1996.

[17] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *proceedings of the 27th Real-Time Systems Symposium (RTSS)*, 2006.

[18] R. Davis and A. Burns. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In *proceedings of the 30th Real-Time Systems Symposium (RTSS)*, pages 398–409, 2009.

[19] R. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):35:1–35:44, October 2011.

[20] M. Dellinger, A. Lindsay, and B. Ravindran. An experimental evaluation of the scalability of real-time scheduling algorithms on large-scale multicore platforms. *Journal of Experimental Algorithmics*, 17:4.3:4.1–4.3:4.22, Oct. 2012.

[21] A. Easwaran, I. Shin, I. Lee, and O. Sokolsky. Bounding preemptions under edf and rm schedulers. Technical report MS-CIS-06-07, University of Pennsylvania, 2013.

[22] K. Jeffay and D. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *proceedings of the 14th Real-Time Systems Symposium (RTSS)*, 1993.

[23] D. I. Katcher, H. Arakawa, and J. K. Strosnider. Engineering and analysis of fixed priority schedulers. *IEEE Transactions on Software Engineering*, 19(9):920–934, 1993.

[24] S. Kato and N. Yamasaki. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS)*, 2009.

[25] J. Lelli, D. Faggioli, T. Cucinotta, and G. Lipari. An experimental comparison of different real-time schedulers on multicore systems. *Journal of Systems and Software*, 85(10):2405–2416, 2012.

[26] L. Leyva-del-Foyo, P. Mejia-Alvarez, and D. de Niz. Predictable interrupt management for real time kernels over conventional PC hardware. In *proceedings of the 12th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2006.

[27] W. Lunniss, S. Altmeyer, and R. Davis. A Comparison between Fixed Priority and EDF Scheduling accounting for Cache Related Pre-emption Delays. *Dagstuhl Transactions on Embedded Systems (LITES)*, 1(1):01:1–01:24, 2014.

[28] W. Lunniss, S. Altmeyer, C. Maiza, and R. Davis. Integrating cache related pre-emption delay analysis into EDF scheduling. In *proceedings of the 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[29] R. Rajkumar, L. Sha, and J.P. Lehoczky. On countering the effects of cycle-stealing in a hard real-time environment. In *proceedings of the 8th Real-Time Systems Symposium (RTSS)*, pages 2–11, 1987.

[30] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemptions. *ACM Transactions on Embedded Computing Systems*, 10(2):27:1–27:34, December 2010.

[31] P. Sousa, K. Bletsas, B. Andersson, and E. Tovar. Practical aspects of slot-based task-splitting dispatching in its schedulability analysis. In *proceedings of the 17th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 224–230, 2011.

[32] P. Sousa, K. Bletsas, E. Tovar, P. Souto, and B. Akesson. Unified overhead-aware schedulability analysis for slot-based task-splitting. *Real-Time Systems*, 50(5-6):680–735, 2014.

[33] P. Sousa, P. Souto, E. Tovar, and K. Bletsas. The Carousel-EDF scheduling algorithm for multiprocessor systems. In *proceedings of the 19th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2013.

[34] M. Spuri. Analysis of deadline scheduled real-time systems. Technical report, INRIA, 1996.

[35] Z. Wu, Y. Krish, and R. Pellizzoni. Worst case analysis of DRAM latency in multi-requestor systems. In *proceedings of the 34th Real-Time Systems Symposium (RTSS)*, 2013.

[36] F. Zhang and A. Burns. Schedulability analysis for real-time systems with edf scheduling. *IEEE Transactions on Computers*, 58(9):1250–1258, 2009.

[37] F. Zhang and A. Burns. Schedulability analysis of edf-scheduled embedded real-time systems with resource sharing. *ACM Transactions on Embedded Computing Systems*, 12(3):67:1–67:19, March 2013.

APPENDIX A.

The use of a global clock allows us to reduce the jitter of the release of a remote subtask and of the IPI handler. In the following subsections we provide a detailed analysis.

Without the use of a global clock the release of a migratory subtask would have to be done by means of an IPI sent at the end of the previous subtask, as appears to be used by Bastoni et al. [10] in their EDF-WM implementation. This has two negative effects. First, the IPI latency is now on the critical path, and therefore needs to be fully accounted in the jitter of the release of a migratory subtask, rather than just the jitter in the IPI latency as in the implementation outlined in this paper. Second, this overhead needs to be added for each subtask, so that the release jitter of the last subtask of a task split into a large number of subtasks is much larger. In a global-clock based implementation, all migratory subtasks have the same jitter, which does not account for the full IPI latency but only its jitter.

A. Release jitter of a remote subtask

In the release of a remote subtask, we can use the global clock to reduce the jitter it experiences. The key observation is that, independent of the delay in processing the arrival of the first subtask, we may compute an upper bound on the length of the time interval since the release of the first subtask until the completion of any subtask. Therefore, when the (first sub)task is released, the global clock is read, and an IPI is sent to the processors running other subtasks of the same task. Upon processing the IPI at these processors, the relevant interrupt handler can program a timer to release a subtask at the appropriate global time, based on the relative delay by which completion of the previous subtask is guaranteed to have occurred and the global time read when the task was first released.

Thus, the release of either a middle or a last subtask suffers an additional jitter, with respect to the jitter of the migratory task, J_s, comprising the response time $RihR_s$ of the release interrupt handler of the first subtask and the precision, π of the reading of the global clock. Therefore, J_i in (II-B) and (5) for all subtasks, τ_s^i, that are not the first of the respective migratory task is replaced by:

$$J_s^i = J_s + RihR_s + \pi$$

$RihR_s$ comprises a blocking term $IntB_p$, caused by disabling interrupts and preemptions (note p is the processor of the first subtask), and another term $IntC_p$, caused by the contention among the different interrupt sources. $RihR_s$ is given by:

$$RihR_s = IntB_p + IntC_p \tag{19}$$

In our model, blocking of interrupts may occur either when a task is running, IpB, or when the kernel executes the scheduler and/or the context switch. The cost of the latter depends on the events that trigger the execution of the scheduler and/or the context switch, as described in Sec. III. From that description we derive:

$$IntB_p = \begin{cases} max(IpB, SchedO + TsetO + MigrO), \text{ if } p \text{ has} \\ \qquad \text{some other subtask that is not the last} \\ max(IpB, SchedO + TsetO), \text{ otherwise} \end{cases}$$

If there is another subtask that is not a last subtask, then the maximum blocking may occur upon the context switch due to the expiration of the budget enforcement timer of such a task, which incurs an additional $MigrO$ cost, with respect to context switches triggered by other events, such as the release of another job.

The occurrence of multiple interrupts at approximately same time may add to the delay in handling a particular interrupt. In our model, we consider three types of interrupts: interrupts that release the job of a task, IPIs that are used in the release of remote subtasks and the interrupts generated by budget enforcement timers. Each of these interrupts is associated with a particular task, and we assume that in any continuous time interval in which there is some interrupt being handled or pending in processor p, there is at most one interrupt per (sub)task in p. Thus $IntD_p$ is given by:

$$IntC_p = |p| \cdot max(RelO + TsetO, IpiO, BetO)$$

where $|p|$ is the number of (sub)tasks in processor p.

If the migratory task is periodic, it is possible to reduce even more the jitter. The key observation is that, in that case, the global time at which the release interrupt of the (first sub)task is raised is known. From the relative deadline

of the each subtask, it is possible to derive the global time by which each subtask will have completed. Thus the IPI handler can program a local timer to expire at that global time, and the jitter of any subtask can be reduced to:

$$J_s^i = J_s + \pi$$

where J_s is the jitter of the migratory task and π is the precision of the global clock. The latter arises because when the global clock is read in the IPI handler there is an uncertainty of π with respect to actual value of the global clock.

B. Jitter of the IPI Handler

The blocking of interrupts and the possibility of interrupt contention at the processor of the first subtask may introduce variable delays in the sending of IPI to the processors of the other subtasks, and thus to some jitter that adds to the jitter of the (first sub)task and the IPI jitter proper. Because, the sending of the IPI occurs at the end of the release of the first subtask, we use the release interrupt handler response time, $RihR_s$, of the (first sub)task for this additional jitter. That is, the interference caused by the IPI handler for each subtask is given by:

$$IpiI_s^i(t) = \left\lceil \frac{t + J_s + RihR_s + IpiJ}{T_s} \right\rceil \cdot IpiO$$

where $RihR_s$ is given by (19).

APPENDIX B.

We now describe how to adapt the partitioning and splitting algorithm presented in the original EDF-WM paper [24], so as to apply the overhead-aware analysis developed in section III. The main difference is that we use QPA sensitivity analysis to compute the largest value that each processor can accommodate for the migratory task.

As in the original algorithm, we first try to assign every task as a non-migratory task. If a task cannot be assigned as non-migratory, the number of parts in which it is split is determined tentatively, starting with two and incrementing it by one until either the task is successfully assigned or the number of parts exceeds the number of processors in the system.

Because the overheads of each type of subtask are different, we consider each type separately. For each tentative number of processor where the task may run, s, we start with the first subtask and use (14) and QPA-based sensitivity analysis to compute, for every processor, the largest value C_s^1 that preserves its schedulability, assuming $D_s^1 = D/s$. The processor with the largest computed C_s^1 is chosen to run the first subtask. Next, we apply QPA-based sensitivity analysis to compute, for each of the remaining processors, the largest value C_s^m that preserves its schedulability. The processors for the 2nd, 3rd, ... $(s-1)$-th subtasks are chosen in order of non-decreasing values of C_s^m. Finally, for the last subtask, we need to check, using (14) and QPA, whether its computation demand $(C_s - \sum_i C_s^i)$ can be accommodated on its respective processor. Accordingly, either the last subtask is assigned to that processor or else, the migratory task is unschedulable as s subtasks and we need increment the tentative number of subtasks by one and retry. Since this affects the subtask deadline D/s, the respective C_s^i needs to be recomputed. Eventually, either the task is split successfully or, if this is not possible even when using all m processors, the task set is deemed unschedulable.

Our implementation of QPA-based sensitivity analysis relies on binary-search to compute both C'^1_s and C'^m_s rather than C_s^1 or C_s^m. The latter are computed from the former using (9) and (10), respectively. The initial interval is set to $[0, min(C'_s, D_s/s)]$, because splitting is attempted only if C'_s cannot be accommodated on a single processor. In each iteration of the binary search, we set C'^i_s to the mid-point of the current interval and apply QPA using (14). Depending on whether the task set is deemed schedulable or not, we set respectively the lower-end or the upper-end of the interval to the previous mid-point. When the width of the interval is 1, we assign C'^i_s the value of the lower end of the interval and terminate.

Note that in QPA, schedulability must be checked only at the deadlines, otherwise, a taskset that is schedulable may fail QPA. This is because we account for $RelO$ and $IpiO$ at the earliest time these overheads may be incurred, not at the deadline of the respective task.

In EDF-WM, QPA-based sensitivity analysis may lead to backtracking. To understand why, assume that a first subtask, τ_s^1 is assigned to a processor p. The mapping of the other subtasks of τ_s to processors, that follows immediately, takes into account the number of (sub)tasks on p, according to the analysis in Sec. III (cf. (13)). Further assume that after this mapping, EDF-WM assigns another (sub)task to p, incrementing the number of (sub)tasks in p. This invalidates the schedulability analysis of the processors assigned the other subtasks of τ_s. Therefore, a new schedulability analysis with the new number of (sub)tasks in p must be performed for each of these processors. If some processor fails this analysis, we need to backtrack the assignment of the tasks up to τ_s.

Note that for C=D, QPA sensitivity analysis does not require backtracking. The reason is that all first subtasks are notionally C=D, and therefore once the algorithm assigns a first subtask to a processor, no other (sub)task will be assigned to that processor.