An Overview of Interrupt Accounting Techniques for Multiprocessor Real-Time Systems*

Björn B. Brandenburg Hennadiy Leontyev James H. Anderson

Abstract

The importance of accounting for interrupts in multiprocessor real-time schedulability analysis is discussed and three interrupt accounting methods, namely quantum-centric, task-centric, and processor-centric accounting, are analyzed and contrasted. Additionally, two special cases, dedicated interrupt handling (*i.e.*, all interrupts are processed by one processor) and timer multiplexing (*i.e.*, all jobs are released by a single hardware timer), are considered and corresponding analysis is derived. All discussed approaches are evaluated in terms of schedulability based on interrupt costs previously measured on a Sun Niagara multicore processor. The results show that there is no single "best" accounting technique that is always preferable, but rather that the relative performance of each approach varies significantly based on task set composition, *i.e.*, the number of tasks and the maximum utilization.

1 Introduction

System overheads such as time lost to task switches and scheduling decisions must be accounted for in real-time systems if temporal correctness is to be guaranteed (Cofer and Rangarajan, 2002; Liu, 2000). Of these overheads, *interrupts* are notoriously troublesome for real-time systems since they are not subject to scheduling and can significantly delay real-time tasks.

*Work supported by IBM, Intel, and Sun Corps.; NSF grants CNS 0834270, CNS 0834132, and CNS 0615197; and ARO grant W911NF-06-1-0425.

In work on uniprocessor real-time systems, methods have been developed to account for interrupts under the two most-commonly considered real-time scheduling policies (Liu and Layland, 1973): under *static-priority* scheduling, interrupts can be analyzed as higher-priority tasks (Liu, 2000), and under *earliest-deadline-first* (EDF) scheduling, schedulability can be tested by treating time lost to processing interrupts as a source of blocking (Jeffay and Stone, 1993).

Properly—but not too pessimistically—accounting for interrupts is even more crucial in multiprocessor real-time systems. Due to their increased processing capacity, such systems are likely to support much higher task counts, and since real-time tasks are usually invoked in response to interrupts, multiprocessor systems are likely to service interrupts much more frequently. Further, systematic pessimism in the analysis has a much larger impact on multiprocessors, as is shown below.

Unfortunately, interrupts have not received sufficient attention in work on multiprocessor real-time systems. The first and, to the best of our knowledge, only published approach to date was proposed by Devi (2006). Devi presented a *quantum-centric* accounting method in which the length of the system's scheduling quantum is reduced to reflect time lost to overheads. In this paper, we consider this method, as well as two others, in the context of global scheduling algorithms. When the quantum-centric method is applied in this context, it is usually necessary to assume that all possible interrupts occur on all processors in every quantum. This assumption is obviously quite pessimistic and motivates the consideration of other approaches.

Motivating example. In a recent case study on a 32-processor platform involving up to 600 light[1] tasks, the release overhead (*i.e.*, the time taken to process a timer interrupt and invoke a real-time task) of a *global EDF* (G-EDF) implementation was measured to exceed 50μs in the worst case (Brandenburg et al., 2008). Given the system's quantum size of 1000μs, the quantum-centric method would have deemed any task set of 20 or more tasks unschedulable—with fewer tasks than processors, this is clearly excessively pessimistic.

In the above case study, a new, less pessimistic "task-centric" accounting method (see Section 4)

[1]The utilization of the tasks was distributed uniformly in $[0.001, 0.1]$. Please see Section 6.1 and (Brandenburg et al., 2008) for a detailed description of these experiments.

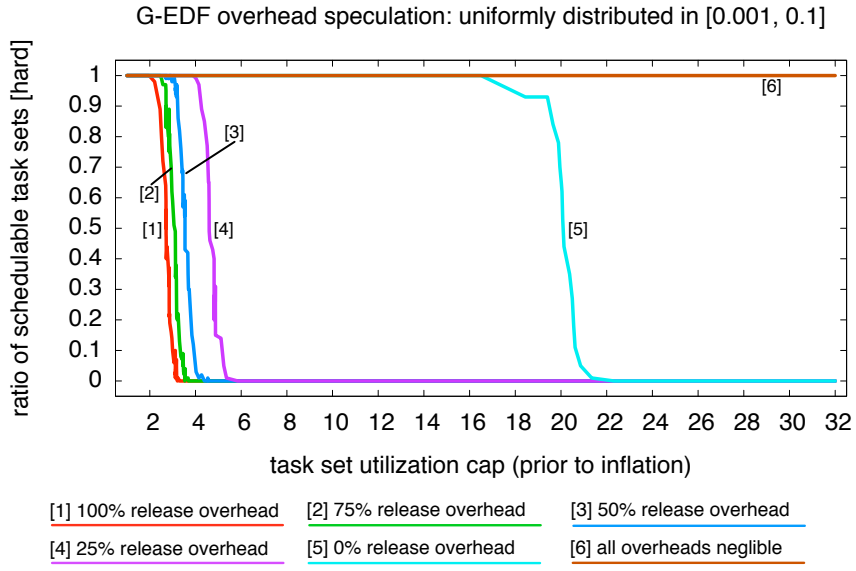

Figure 1: Hard real-time schedulability under G-EDF on a 32-processor platform assuming reduced release overhead, which is accounted for using the task-centric method. Note that all task sets are schedulable if all overheads are assumed to be negligible. (This graph corresponds to Figure 2(a) in (Brandenburg et al., 2008); see Section 6 for details.)

was developed. However, even with "task-centric" accounting, G-EDF performed worse than expected. Suspecting high job release overhead to be the cause, we conducted simulations to estimate performance assuming reduced overhead. Surprisingly, we found that even with a 75% reduction in release overhead, schedulability increases very little (see Figure 1). However, the experiments also confirmed release overhead as the leading cause of reduced schedulability—performance improved dramatically assuming overhead-free releases. This discrepancy stems from quadratically-growing pessimism in the "task-centric method," as is shown in Section 4.3.3. Figure 1 highlights that accurate accounting for overheads is crucial to multiprocessor real-time performance, and that systematic pessimism can have a significant impact on multiprocessor systems due to high task counts.

Contributions. This paper provides an overview of all currently-known multiprocessor interrupt accounting techniques for global schedulers. In particular, we

- highlight the importance of accurate interrupt accounting for multiprocessor real-time systems and survey commonly-encountered interrupt sources in current multiprocessor systems

(Section 2, formalized in Section 3);

- summarize "quantum-centric" interrupt accounting (Section 4.1) and describe "task-centric" interrupt accounting (Section 4.3), which has been used—but not described in detail—in previous studies (Brandenburg et al., 2008; Brandenburg and Anderson, 2008, 2009) to overcome the limitations of "quantum-centric" accounting, and show that the "task-centric" method still over-estimates interrupt delays by a factor that is quadratic in the number of tasks (Section 4.3.3);

- show how interrupt accounting is fundamentally different from previous work on reduced-capacity scheduling and, by addressing these differences, propose "processor-centric" interrupt accounting, which is designed to overcome the pessimism inherent in "task-centric" accounting (Section 4.4);

- discuss two common special cases, namely how job releases are delayed if interrupts are handled by a dedicated processor (Section 5.2), and further how said delays change if all job releases are triggered by a single hardware timer (Section 5.3); and

- evaluate the effectiveness of all considered approaches in terms of schedulability for both hard and soft real-time systems and show that none of the proposed approaches strictly dominates the others, *i.e.*, there is no single "best" approach (Section 6).

This paper extends a previous conference version (Brandenburg et al., 2009); in particular, a discussion of delays due to "inter-processor interrupts" (Section 4.2) and common special cases (Section 5) has been added, and the experiments (Section 6) were augmented to include all special cases and repeated with more-recent overhead measurements (Brandenburg and Anderson, 2009), thus reflecting over a year of implementation improvements.

Next, we provide a survey of common interrupt types and describe how they are usually serviced by operating systems.

2 Interrupts

To motivate our system model, we begin by providing a high-level overview of interrupts in a modern multiprocessor architecture. We focus on Intel's x86 architecture because it is in widespread use and well-documented (Intel Corp., 2008a,b), but the discussion similarly applies to other multiprocessor architectures as well (Weaver and Germond, 1994; Saulsbury, 2008).

Interrupts notify processors of asynchronous events and may occur between (almost) any two instructions. If an interrupt is detected, the processor temporarily pauses the execution of the currently-scheduled task and executes a designated *interrupt service routine* (ISR) instead. This can cause the interrupted task to incur undesirable delays that must be accounted for during schedulability analysis.

Most interrupts are *maskable*, *i.e.*, the processor can be instructed by the OS to delay the invocation of ISRs until interrupts are unmasked again. However, *non-maskable interrupts* (NMIs), which can be used for "watch dog" functionality to detect system hangs, cannot be suppressed by the OS (Intel Corp., 2008b).

In multiprocessor systems, some interrupts may be local to a specific processor (*e.g.*, register-based timers (Weaver and Germond, 1994)), whereas others may be serviced by multiple or all processors.

Interrupts differ from normal preemptions in that a task cannot migrate while it is being delayed by an ISR, *i.e.*, a task cannot resume execution on another processor to reduce its delay. This limitation arises due to the way context switching is commonly implemented in OSs. For example, in Linux (and Linux-derived systems such as the one considered in (Brandenburg et al., 2008)), there is only a single function in which context switching can be performed, and it is only invoked at the *end* of an ISR (if a preemption is required). From a software engineering point of view, limiting context switches in this way is desirable because it significantly reduces code complexity. In terms of performance, ISRs tend to be so short (in well-designed systems) that context-switching and migration costs dominate ISR execution times. Hence, delaying tasks is usually preferable to allowing migrations unless either migration and scheduling costs are negligible or ISR execution times are excessively long.

Delays due to ISRs are fundamentally different from scheduling and preemption overheads: the occurrence of scheduling and preemption overheads is controlled by the OS and can be carefully planned to not occur at inopportune times. In contrast, ISRs execute with a statically-higher priority than any real-time task in the system and cannot be scheduled, *i.e.*, while interrupts can be temporarily masked by the OS, they cannot be selectively delayed[2], and thus are not subject to the scheduling policy of the OS.

2.1 Interrupt Categories

Interrupts can be broadly categorized into four classes: *device interrupts* (DIs), *timer interrupts* (TIs), *cycle-stealing interrupts* (CSIs), and *inter-processor interrupts* (IPIs). We briefly discuss the purpose of each next.

DIs are triggered by hardware devices when a timely reaction by the OS is required or to avoid costly "polling" (see below). In real-time systems, DIs may cause jobs to be released, *e.g.*, a sensor may trigger a DI to indicate the availability of newly-acquired data, in response to which a job is released to process said data.

TIs are used by the OS to initiate some action in the future. For example, TIs are used to support high-resolution delays ("sleeping") in Linux. They can also be used for periodic job releases and to enforce execution time budgets. In networking protocols, TIs are commonly employed to trigger timeouts and packet re-transmissions.

CSIs are an artifact of the way that modern hardware architectures are commonly implemented and differ from the other categories in that they are neither controlled nor handled by the OS. CSIs are used to "steal" processing time for some component that is—from the point of view of the OS—hardware, but that is implemented as a combination of hardware and software (so called "firmware") and that lacks its own processor. CSIs are intended to be transparent from a logical correctness point of view, but of course do affect temporal correctness.[3] They are usually non-

[2]Masking may create non-trivial timing dependencies because it usually affects physical *interrupt lines*, which are oftentimes shared among multiple interrupt sources.

[3]CSIs are especially problematic if the code that is being executed is unknown—for example, a CSI could flush instruction and data caches and thereby unexpectedly increase task execution costs.

maskable and the OS is generally unaware if and when CSIs occur. A well-known example for the use of CSIs is the system management mode (SMM) in Intel's x86 architecture (Intel Corp., 2008a,b): when a system management interrupt (SMI) occurs, the system switches into the SMM to execute ISRs stored in firmware. For example, on some chip sets the SMM is entered to control the speed of fans for cooling purposes. CSIs can also occur in architectures in which raw hardware access is mediated by a hypervisor (such as Sony's PlayStation 3 (Kurzak et al., 2008) and SUN's sun4v architecture (Saulsbury, 2008))—the hypervisor may become active at any time to handle interrupts or perform services for devices "invisible" to the OS.

In contrast to DIs, TIs, and CSIs, the final category considered, IPIs, are specific to multiprocessor systems. IPIs are used to synchronize state changes across processors and are generated by the OS. For example, the modification of memory mappings (*i.e.*, changes to address spaces) on one processor can require software-initiated TLB flushes on multiple processors (Intel Corp., 2008b,c). IPIs are also commonly used to cause a remote processor to reschedule.

2.2 Delay Avoidance

There are four implementation choices that help to limit interrupt-related delays: split interrupt handling, polling, interrupt masking, and, in multiprocessor systems, dedicating a processor to interrupt handling.

Split interrupt handling can be used to reduce the length of ISRs. With split interrupt handling, the work required to handle an interrupt is divided into two parts: a short ISR only acknowledges the interrupt and does the minimum amount of work necessary for correct operation, whereas the main work is carried out by an interrupt thread that is subject to OS scheduling (Liu, 2000). However, even with split interrupt handling, some work, such as releasing jobs or activating interrupt threads, must be carried out in the ISRs themselves, and this work must be accounted for.

DIs can be avoided altogether through *polling*, whereby hardware devices are probed periodically for state changes and pending events. However, TIs are still required to invoke the scheduler and initiate polling periodically.

In embedded systems that execute real-time tasks in privileged mode, interrupts can be *masked*

whenever a real-time job is executing. Once a job completes, interrupts are unmasked and pending interrupts handled. While this helps to make interrupts more predictable, it does not reduce the time lost to ISRs. Further, this approach does not apply to NMIs and, for security and safety reasons, it is highly undesirable to run real-time tasks in privileged mode. Additionally, if timing constraints are stringent, then the associated increase in interrupt latency may be prohibitive. If IPIs are masked for prolonged times, then concurrency may be reduced while processors wait to synchronize.

Both polling and masking interrupts for prolonged times can drastically reduce the maximum I/O throughput by increasing device idle time. Such throughput losses may be economically undesirable or even unacceptable (especially in soft real-time systems). Finally, neither approach is a viable choice in the increasingly-relevant class of general purpose operating systems with real-time properties (such as real-time variants of Linux). Further, if a hardware platform makes use of CSIs, then, by definition, they cannot be avoided by any OS design.

With *dedicated interrupt handling* (Stankovic and Ramamritham, 1991), a processor that does not schedule real-time tasks is reserved for servicing all DIs and most[4] TIs. Thus, real-time applications are *shielded* from interrupts other than CSIs and IPIs and hence incur fewer delays. However, such a design may lead to an increased reliance on IPIs since interrupt-triggered scheduling decisions must be relayed from the dedicated processor to the processors on which real-time applications execute. Such IPIs can increase release latencies and must be accounted for in schedulability analysis.

In practice, dedicated interrupt handling is commonly used. Variants of dedicated interrupt handling have been developed for several real-time Linux variants (Brosky and Rotolo, 2003; Piel et al., 2005; Betti et al., 2008); dedicated interrupt handling is also supported in recent standard versions of Linux.

To summarize, interrupts can delay real-time tasks and cannot be avoided completely on most (if not all) modern multiprocessor architectures; hence, they must be accounted for in schedulability analysis.

[4]Depending on the particular implementation, periodic timer ticks may still be serviced by all processors. For example, this is the case with standard Linux.

2.3 Bounding Interrupt Interference

Bounding ISR activations may be difficult in practice. The number of distinct interrupts is often limited (to reduce hardware costs), and hence interrupts may be shared among devices. Further, many devices and interrupts are commonly multiplexed among many logical tasks. For example, a single timer (and its corresponding ISR) is likely to be shared among multiple real-time tasks and potentially even best-effort tasks.

As a result, it may be impractical to characterize a system's worst-case interrupt behavior by modeling the individual (hardware) interrupts. Instead, it may be more illuminating to consider logical "interrupt sources" that cause one or more ISRs to be invoked in some pattern, but do not necessarily correspond to any particular device. This approach is formalized in the next section.

3 System Model

We consider the problem of scheduling a set of n implicit-deadline sporadic tasks $\tau = \{T_1, \ldots, T_n\}$ on m processors; we let $T_i(e_i, p_i)$ denote a task where e_i is T_i's *worst-case per-job execution time* and p_i is its *period*. $T_{i,j}$ denotes the j^{th} job ($j \geq 1$) of T_i. $T_{i,j}$ is released at time $r_{i,j} \geq 0$ and should complete by its absolute deadline $d_{i,j} = r_{i,j} + p_i$. If $j > 1$, then $r_{i,j} \geq r_{i,j-1} + p_i$. If $T_{i,j}$ completes at time t, then its *tardiness* is $\max(0, t - d_{i,j})$. A task's tardiness is the maximum of the tardiness of any of its jobs. Note that, even if $T_{i,j}$ misses its deadline, $r_{i,j+1}$ is not altered. However, tasks are sequential: $T_{i,j+1}$ cannot start execution until $T_{i,j}$ completes. T_i's *utilization* is $u_i = e_i/p_i$; τ's *total utilization* is $U(\tau) = \sum_{i=1}^{n} u_i$. We assume $U(\tau) \leq m$; otherwise, tardiness may grow unboundedly (Devi, 2006).

Scheduler. In this paper, we assume preemptive G-EDF scheduling (*i.e.*, jobs with smaller $d_{i,j}$ values have higher priority). In an *event-driven* system, the scheduler is invoked whenever a job is released (to check if a preemption is required) or completes (to select the next job, if any). In contrast, in a *quantum-driven* system, the scheduler is invoked only at integer multiples of a *scheduling quantum* Q (see (Devi, 2006; Liu, 2000) for overviews). Hence, job releases and completions may

be processed with a delay of up to Q time units[5] and all task parameters must be integer multiples of Q.

Interrupts. An *interrupt source* causes ISRs to be invoked. An interrupt source is *local* if all invoked ISRs are serviced on the same processor, and *global* otherwise. We consider a system with r global interrupt sources $I_1, \ldots, I_r$, and, on each processor h, where $1 \leq h \leq m$, r_h local interrupt sources $I_1^h, \ldots, I_{r_h}^h$.

When an ISR is invoked on a processor, the job currently running on that processor is temporarily *stopped* and its completion is delayed. In contrast to a regular preemption, a stopped job cannot migrate to another processor while the interfering ISR executes.

Definition 1. We assume that for each interrupt source I_x (either global or local) there is a monotonic function $\mathsf{dbf}(I_x, \Delta)$ that bounds the maximum service time required by all ISRs invoked by I_x over an interval of length $\Delta \geq 0$.

Additionally, we assume that, for each interrupt source I_x (either local or global), $\mathsf{dbf}(I_x, \Delta)$ is upper-bounded by a linear function of Δ, *i.e.*, there exist $P(I_x) \leq 1$ and $R(I_x) \geq 0$ such that (1) below holds for all $\Delta \geq 0$.

$$\mathsf{dbf}(I_x, \Delta) \leq P(I_x) \cdot \Delta + R(I_x) \tag{1}$$

As an example, for a *sporadic interrupt source* I_x that invokes an ISR of maximum length c_x at most once every p_x time units, the demand bound function is given by

$$\mathsf{dbf}(I_x, \Delta) = \left\lfloor \frac{\Delta}{p_x} \right\rfloor \cdot c_x + \min\left(c_x,\ \Delta - \left\lfloor \frac{\Delta}{p_x} \right\rfloor \cdot p_x\right), \tag{2}$$

as illustrated in Figure 2, in which case, $P(I_x) = \frac{c_x}{p_x}$ and $R(I_x) = c_x$.

We require that, in total, the time spent on interrupt handling does not exceed the capacity of one processor for sufficiently long time intervals. This assumption holds if

$$\sum_{k=1}^{r} P(I_k) + \sum_{h=1}^{m} \sum_{k=1}^{r_h} P(I_k^h) < 1. \tag{3}$$

[5]This delay can be accounted for by shortening a task's period (and hence relative deadline) by Q time units. A choice of $Q = 1$ms is common (Devi, 2006).

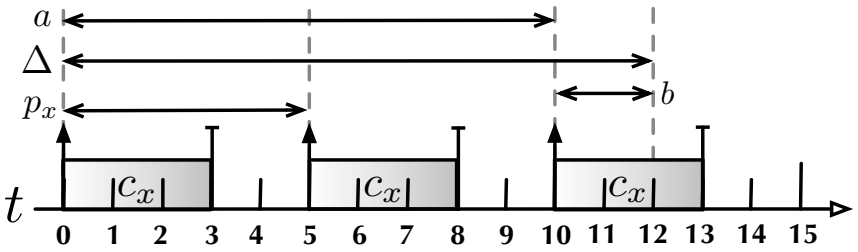

Figure 2: Illustration of (2) for $c_x = 3$, $p_x = 5$, and $\Delta = 12$, where $a = p_x \cdot \left\lfloor \frac{\Delta}{p_x} \right\rfloor$ and $b = \Delta - \left\lfloor \frac{\Delta}{p_x} \right\rfloor \cdot p_x$. At most $\left\lfloor \frac{\Delta}{p_x} \right\rfloor = 2$ complete ISR invocations execute in Δ (times 0–3 and 5–8), and $b = 2$ time units of a third invocation fit within Δ (time 10–12).

Note that systems that violate (3) likely have sufficiently-high processor and task counts to render global scheduling impractical. We expect such large systems to be scheduled using a *clustered* approach (Baker and Baruah, 2007; Calandrino et al., 2007; Brandenburg et al., 2008), wherein (3) applies only on a per-cluster basis (and hence is not a serious limitation).

Schedulability. In a *hard real-time system*, each job must complete by its absolute deadline and, in a *soft real-time system*, each job must have bounded deadline tardiness. We are interested in developing a validation procedure—or *schedulability test*—for determining whether hard or soft real-time constraints are met for a task set τ that is scheduled on m processors using G-EDF in the presence of interrupts.

Many hard and soft real-time schedulability tests for G-EDF without interrupts have been proposed in prior work (Goossens et al., 2003; Baker, 2003; Bertogna et al., 2005; Baruah, 2007; Devi, 2006; Leontyev and Anderson, 2008; Bertogna et al., 2009). In the next section, we describe three methods for incorporating interrupts in existing analysis, Devi's "quantum-centric" method and two recently-developed methods.

4 Schedulability Analysis

Because interrupt handlers effectively have higher priority than ordinary jobs, they may delay job completions. This can be accounted for in three different ways. Under *quantum-centric accounting* (Devi, 2006), interrupts are understood to reduce the "effective quantum length," *i.e.*, the service

time available in each quantum to real-time jobs. Each task's worst-case execution cost is inflated to ensure completion given a lower-bound on the effective quantum length. Under *task-centric accounting*, interrupts are considered to extend each job's actual execution time and worst-case execution times are inflated accordingly. Under *processor-centric accounting*, task parameters remain unchanged and interrupts are considered to reduce the processing capacity available to tasks.

The first two methods are not G-EDF-specific—a task system is deemed schedulable if it passes an existing *sustainable* (Baruah and Burns, 2006) schedulability test assuming inflated worst-case execution times for all tasks.

Processor-centric accounting requires a two-step process: **(i)** reduce the processing capacity of the platform by accounting for the time consumed by interrupt handlers in the worst case; and **(ii)** analyze the schedulability of the original task set on the reduced-capacity platform. While general in principle, the reduced-capacity analysis used in Step (ii) has been developed only for a limited number of scheduling algorithms to date, G-EDF prominently among them (Leontyev and Anderson, 2007; Shin et al., 2008).

4.1 Quantum-Centric Accounting

Recall from Section 3 that under quantum-based scheduling all task parameters are multiples of a quantum size Q and that scheduling decisions are only made at multiples of Q. In practice, some processor time is lost due to system overheads during each quantum; the remaining time is called the *effective quantum length* Q'. With Devi's quantum-based accounting method (Devi, 2006), Q' is derived by assuming that all interrupt sources require maximum service time in each quantum, *i.e.*, on processor h,

$$Q'_h = Q - \sum_{k=1}^{r_h} \mathsf{dbf}(I_k^h, Q) - \sum_{k=1}^{r} \mathsf{dbf}(I_k, Q). \tag{4}$$

Under G-EDF, it generally cannot be predicted on which processor(s) a job will execute, hence the system-wide effective quantum length is the minimum of the per-processor effective quantum lengths, *i.e.*, $Q' = \min\{Q'_h\}$. If $Q' > 0$, then task T_i's inflated worst-case execution time e'_i is given

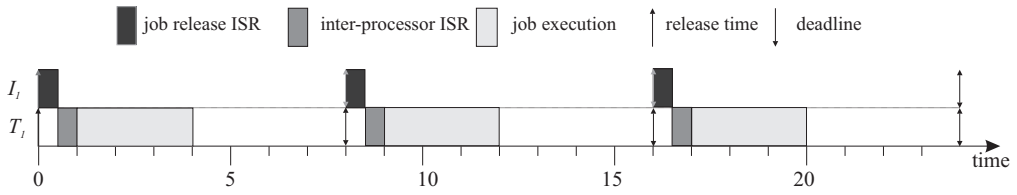

Figure 3: Usage of IPIs for rescheduling in Example 1.

by

$$e'_i = Q \cdot \left\lceil \frac{e_i}{Q'} \right\rceil . \tag{5}$$

If $Q' \leq 0$ or $e'_i > p_i$ for some T_i, then τ is deemed unschedulable. This technique does not depend on the scheduling algorithm in use—in fact, it has been used by Devi to analyze a range of algorithms (Devi, 2006). A major limitation of the quantum-centric method is that it tends to overestimate interrupt frequencies due to the short analysis interval Q. This is especially a concern on large multiprocessor platforms due to the (likely) high number of interrupt sources.

4.2 IPI Delay

Before considering the task- and processor-centric methods in detail, we make the following observation. If scheduling is not quantum-based, then a job release is typically triggered by a TI or DI. In this situation, the processor executing the job-release ISR and the processor on which the newly released job is scheduled may be different. To initiate rescheduling, an IPI is sent to the target processor as the example below illustrates.

Example 1. Consider a task set $\tau = \{T_1(3, 8)\}$ scheduled on two processors. Suppose that an ISR serving interrupts from source I_1, which is invoked every 8 time units and executes for 0.5 time unit, releases jobs of T_1. In the schedule shown in Figure 3, job $T_{1,1}$ is released by the ISR that executes within the interval $[0, 0.5)$ on processor 1. At time 0.5, an IPI is sent to processor 2 and thus the IPI service routine executes during the interval $[0.5, 1)$. After that, job $T_{1,1}$ is scheduled on processor 2. Subsequent jobs of T_1 are handled similarly.

In general, if an IPI is sent to some processor, then some newly released job must be scheduled on that processor. The execution of the associated IPI service routine can be treated as the execution

of preempting task T_i. In the description of the task-centric and processor-centric methods below, we assume that the maximum IPI cost is added to the worst-case execution time of each task. This treatment is similar to accounting for scheduling overheads (Liu, 2000; Devi, 2006).

4.3 Task-Centric Accounting

Based on the observation that $T_{i,j}$ is delayed by at most the total duration of ISRs that are invoked while $T_{i,j}$ executes, the task-centric method analyzes the complete interval during which a job $T_{i,j}$ can execute (instead of focusing on an individual quantum). In this paper, we describe this method in conjunction with G-EDF scheduling, but it can be applied similarly to other scheduling algorithms. Since the length of the analysis interval depends on $T_{i,j}$'s tardiness, the task-centric method is more involved in the soft real-time case. We start by first considering the hard real-time case.

4.3.1 Hard Real-Time Schedulability

The key concept behind the task-centric method is that, if jobs of $T_i(e_i, p_i)$ are delayed by at most δ_i, then T_i will meet all of its deadlines if $T_i'(e_i + \delta_i, p_i)$ meets all of its deadlines assuming no delays (Ha and Liu, 1994). Hence, schedulability in the presence of ISR invocations can be checked with an existing schedulability test oblivious to interrupts if a bound on δ_i can be derived.

Definition 2. Let

$$C(\Delta) = \sum_{k=1}^{r} \mathsf{dbf}(I_k, \Delta) + \sum_{h=1}^{m} \sum_{k=1}^{r_h} \mathsf{dbf}(I_k^h, \Delta)$$

denote a bound on the maximum time consumed by local and global ISRs during a time interval of length Δ.

If $T_{i,j}$ completes by its deadline (*i.e.*, it is not tardy), then it can be *directly* delayed by ISRs for at most $C(d_{i,j} - r_{i,j}) = C(p_i)$ time units.[6] However, $T_{i,j}$ can also be *indirectly* delayed by

[6]Note that in upper-bounding δ_i by $C(p_i)$, local interrupts on *all* processors are considered. A tighter bound on δ_i can be derived if T_i is known to migrate at most $\eta < m$ times: only the $\eta + 1$ processors for which $\sum_{k=1}^{r_h} \mathsf{dbf}(I_k^h, \Delta)$ is maximized must be accounted for in this case.

ISRs that were invoked prior to $r_{i,j}$ by "pushing" processor demand of higher-priority jobs into $[r_{i,j}, d_{i,j})$. Both sources of delay can be accounted for by (pessimistically) assuming that *all* jobs are maximally delayed.

Theorem 1. A task system $\tau = \{T_1, \ldots, T_n\}$ is hard real-time schedulable with G-EDF in the presence of interrupts if $\tau' = \{T'_1, \ldots, T'_n\}$, where $e'_i = e_i + C(p_i)$, passes a sustainable (Baruah and Burns, 2006) hard real-time schedulability test.

Proof. Follows from the preceding discussion. □

4.3.2 Soft Real-Time Schedulability

Our notion of soft real-time schedulability requires jobs to have bounded maximum tardiness. If all ISRs have zero cost, then the following holds.

Theorem 2. (Proved in (Devi, 2006).) If $U(\tau) \leq m$, then for each $T_i \in \tau$ there exists $b_i \geq 0$ such that T_i's maximum tardiness under G-EDF is at most b_i (an expression for b_i is given in (Devi, 2006)).

In the absence of interrupts, every job $T_{i,j}$ is known to only execute within $[r_{i,j}, d_{i,j} + b_i)$. However, in contrast to the hard real-time case, the analysis interval changes in the presence of interrupts since inflating e_i also inflates b_i. Thus, an iterative approach such as the following procedure is required to break the cyclic dependency between the tardiness bound and delays due to ISRs in the soft real-time case.

initialize $b'_i := 0$ for all i

do

set $b^o_i := b'_i$ for all i

set $e'_i := e_i + C(p_i + b^o_i)$ for all i

set $\tau' := \{T'_1, \ldots, T'_n\}$ where $T'_i = (e'_i, p_i)$

compute b'_i for all i with Theorem 2 based on τ'

while ($b'_i \neq b^o_i$ **and** $e'_i \leq p_i$) for all i **and** $U(\tau') \leq m$

Theorem 3. If $U(\tau') \leq m$ and $e'_i \leq p_i$ for all $T_i \in \tau$ after the above procedure terminates, then deadline tardiness of each T_i is at most b'_i under G-EDF scheduling in the presence of interrupts.

Proof. Follows from the preceding discussion. □

4.3.3 Over-Estimation of Interrupt Delays

While often less pessimistic than the quantum-centric method, the task-centric method is also likely to over-charge for interrupts. In fact, the cause of the disappointing (lack of significant) performance improvements observed in Figure 1 is utilization loss inherent in the task-centric method. This loss is quadratic in the number of tasks, as shown below.

Definition 3. Let $c_r > 0$ denote the worst-case execution time of a job-releasing ISR, *i.e.*, the worst-case overhead incurred due to a single job release.

Theorem 4. Consider an event-triggered system τ of n sporadic tasks, where each job is released by an interrupt of cost c_r, where $\min\{p_i\} > c_r > 0$ and $\max\{p_i\}$ is bounded by some constant. Under task-centric interrupt accounting, $U(\tau') = U(\tau) + \Omega(n^2)$, where τ' denotes the inflated task set as defined in Theorems 1 and 3 (respectively).

Proof. First, we consider the hard real-time case. By Theorem 1 and Definition 2,

$$e'_i = e_i + C(p_i) = e_i + \sum_{k=1}^{r} \mathsf{dbf}(I_k, p_i) + \sum_{h=1}^{m} \sum_{k=1}^{r_h} \mathsf{dbf}(I_k^h, p_i).$$

Without loss of generality, assume that the only sources of interrupts $I_1, \ldots, I_n$ are sporadic job releases by $T_1, \ldots, T_n$, where I_i corresponds to releases of T_i. Hence $e'_i = e_i + \sum_{k=1}^{n} \mathsf{dbf}(I_k, p_i)$ and thus

$$U(\tau') = U(\tau) + \sum_{i=1}^{n} \frac{\sum_{k=1}^{n} \mathsf{dbf}(I_k, p_i)}{p_i}.$$

By (2) and Definition 3,

$$\mathsf{dbf}(I_k, p_i) = \left\lfloor \frac{p_i}{p_k} \right\rfloor \cdot c_r + \min\left(c_r,\ p_i - \left\lfloor \frac{p_i}{p_k} \right\rfloor \cdot c_r\right).$$

If $p_i < p_k$, then $\mathsf{dbf}(I_k, p_i) = \min(c_r,\ p_i) = c_r$; otherwise, $\mathsf{dbf}(I_k, p_i) \geq \left\lfloor \frac{p_i}{p_k} \right\rfloor \cdot c_r \geq c_r$. Hence, $\mathsf{dbf}(I_k, p_i) \geq c_r$, and therefore

$$U(\tau') \geq U(\tau) + \sum_{i=1}^{n} \frac{n \cdot c_r}{p_i} \geq U(\tau) + \frac{n^2 \cdot c_r}{\max\{p_i\}}.$$

This establishes a lower bound for the hard real-time case. The same lower bound applies in the soft real-time case as well since $C(p_i + b_i) \geq C(p_i)$ due to the monotonicity of $\mathsf{dbf}(I_k, \Delta)$ (recall that, by definition, $b_i \geq 0$). □

If each (of n) tasks releases a constant number of jobs over some interval, then the time lost to job releases over the interval is $O(n)$. Hence, Theorem 4 shows that the task-centric method asymptotically overestimates the time lost to interrupt processing. For sufficiently large n, reducing the release overhead c_r has comparably little impact—that is, optimizing the OS implementation does not help to mitigate asymptotic inefficiencies in the analysis. Next, we discuss the remaining processor-centric interrupt accounting method, which is designed to overcome this problem.

4.4 Processor-Centric Accounting

The final method, which involves subtracting the ISR time from the total available processing capacity, introduces several difficulties.

First, even though an ISR makes a processor unavailable to ordinary real-time jobs, the currently-running job cannot migrate to another processor. This differs from how limited processing capacity has been treated in prior work (Leontyev and Anderson, 2007; Shin et al., 2008; Leontyev et al., 2009).

Second, if a job is released as a result of an ISR, then it cannot be scheduled on any processor before the respective ISR completes. Both aspects are illustrated by the following example, in which tardiness may grow unboundedly even though tardiness is deemed bounded using traditional reduced-capacity analysis (Leontyev and Anderson, 2007).

Example 2. Consider the task set $\tau = \{T_1(999, 1000)\}$ scheduled on two processors. Suppose that the only ISR in the system, which releases T_1's jobs, is invoked every 1000 time units and

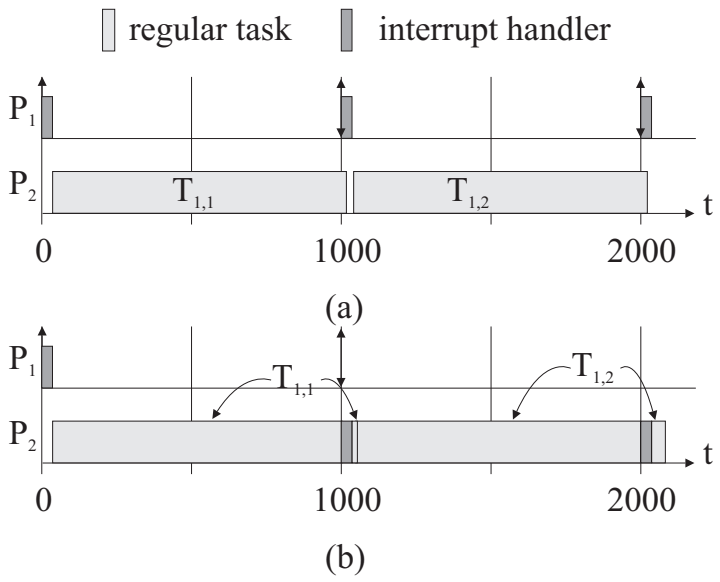

Figure 4: Interrupt handling scenarios from Example 2.

executes for at most 2 time units. In the schedule shown in Figure 4(a), job $T_{1,1}$, released at time 0, is not available for scheduling before time 2. At time 2, $T_{1,1}$ is scheduled on processor 2 and completes at time 1001, so its tardiness is 1. At time 1000, job $T_{1,2}$ arrives and the ISR is invoked on processor 1 so job $T_{1,2}$ becomes available to the scheduler at time 1002 and completes at time 2001. Also, running jobs of task T_1 are not paused in this schedule. In contrast to this, in the schedule in Figure 4(b), the second interrupt is handled by processor 2 and thus preempts job $T_{1,1}$, rendering both processors unavailable to T_1. If the ISR is invoked on the processor that schedules a job of T_1, then the tardiness will grow unboundedly because only 998 execution units are available to T_1 while it requests 999 time units every 1000 time units (assuming that future jobs are released periodically).

In order to re-use results for platforms with limited processor availability, we assume that, when an interrupt occurs, *all* processors become unavailable to jobs in τ for the duration of an ISR. While pessimistic, the above example illustrates that this global capacity reduction is required in order to achieve general applicability of the analysis. We next introduce some definitions to reason about platforms with limited processing capacity.

Definition 4. Let $\mathsf{supply}_h(t, \Delta)$ be the total amount of processor time available on processor h to

the tasks in τ during the interval $[t, t + \Delta)$.

Definition 5. To deal with limited processor supply, the notion of *service functions* has been proposed (Chakraborty et al., 2006). The service function $\beta_h^l(\Delta)$ lower-bounds $\mathsf{supply}_h(t, \Delta)$ for each time $t \geq 0$. We require

$$\beta_h^l(\Delta) \geq \max(0, \widehat{u_h} \cdot (\Delta - \sigma_h)), \tag{6}$$

for $\widehat{u_h} \in (0, 1]$ and $\sigma_h \geq 0$.

In the above definition, the superscript l stands for "lower bound," $\widehat{u_h}$ is the total long-term utilization available to the tasks in τ on processor h, and σ_h is the maximum length of time when the processor can be unavailable. Note that, if processor h is fully available to the tasks in τ, then $\beta_h^l(\Delta) = \Delta$.

Some schedulability tests for limited processing capacity platforms require that the total time available on *all* processors be known (Shin et al., 2008).

Definition 6. Let $\mathsf{Supply}(t, \Delta) = \sum_{h=1}^{m} \mathsf{supply}_h(t, \Delta)$ be the cumulative processor supply during the interval $[t, t + \Delta)$. Let $\mathcal{B}(\Delta)$ be the guaranteed total time that all processors can provide to the tasks in τ during any time interval of length $\Delta \geq 0$.

If lower bounds on individual processor supply are known, then we can compute a lower bound on the total supply using the following trivial claim.

Claim 1. If individual processor service functions $\beta_h^l(\Delta)$ are known, then $\mathsf{Supply}(t, \Delta) \geq \mathcal{B}(\Delta)$, where $\mathcal{B}(\Delta) = \sum_{h=1}^{m} \beta_h^l(\Delta)$.

In the remainder of this section, we assume that individual processor service functions are known. We will derive expressions for them later in Section 4.5.

Hard real-time schedulability of τ on a platform with limited supply can be checked using results from (Shin et al., 2008). This paper presents a sufficient pseudo-polynomial test that checks whether the time between any job's release time and its completion time does not exceed a predefined bound. This test involves calculating the minimum guaranteed supply for different values of Δ.

To test soft real-time schedulability of τ on a platform with limited supply we use the following definition and theorem.

Definition 7. Let $U_L(y)$ be the sum of $\min(|\tau|, y)$ largest task utilizations.

Theorem 5. (Proved in (Leontyev and Anderson, 2007)) Tasks in τ have bounded deadline tardiness if the following inequalities hold

$$U(\tau) \leq \sum_{h=1}^{m} \widehat{u_h} \tag{7}$$

$$\sum_{h=1}^{m} \widehat{u_h} > \max(H-1, 0) \cdot \max(u_i) + U_L(m-1), \tag{8}$$

where H is the number of processors for which $\beta_h^l(\Delta) \neq \Delta$.

In the above theorem, if (7) does not hold, then the long-term execution requirement for tasks will exceed the total long-term guaranteed supply, and hence, the system will be overloaded. On the other hand, (8) implicitly restricts the maximum per-task utilization in τ due to the term $\max(H - 1, 0) \cdot \max(u_i)$, which could be large if the maximum per-task utilization is large. This is especially the case if all processors can be unavailable to tasks in τ, *i.e.*, $H = m$. Since we assumed that all processors are unavailable to τ for the duration of an ISR, this may result in pessimistically claiming a system with large per-task utilizations to be unschedulable. In the next section, in Example 3, we show that such a penalty may be unavoidable.

4.5 Deriving β Service Functions

We now establish a lower bound on the supply provided by processor h to τ over an interval of length Δ.

Definition 8. Let $F = \sum_{k=1}^{r} P(I_k) + \sum_{h=1}^{m} \sum_{k=1}^{r_h} P(I_k^h)$ and $G = \sum_{k=1}^{r} R(I_k) + \sum_{h=1}^{m} \sum_{k=1}^{r_h} R(I_k^h)$.

Lemma 1. If interrupts are present in the system, then any processor h can supply at least

$$\beta_h^l(\Delta) = \max(0, \Delta - C(\Delta)) \tag{9}$$

time units to the tasks in τ over any interval of length Δ. Additionally, (6) holds for $\widehat{u_h} = 1 - F$ and $\sigma_h = \frac{G}{1-F}$.

Proof. We first prove (9). By Definition 4, processor h provides $\mathsf{supply}_h(t, \Delta)$ time units to τ during the interval $[t, t + \Delta)$. By our assumption, processor h is unavailable for the total duration of all local and global ISRs invoked during the interval $[t, t + \Delta)$. From this and Definition 2, we have

$$\mathsf{supply}_h(t, \Delta) \geq \Delta - C(\Delta)$$

Because $\mathsf{supply}_h(t, \Delta)$ cannot be negative, (9) follows from Definition 5. Our remaining proof obligation is to find $\widehat{u_h}$ and σ_h such that (6) holds. From (9), we have

$$\begin{aligned}
\beta_h^l(\Delta) = &\max(0, \Delta - C(\Delta)) \\
&\{\text{by Definition 2}\} \\
\geq &\max\left(0, \Delta - \left[\sum_{k=1}^{r} \mathsf{dbf}(I_k) + \sum_{h=1}^{m}\sum_{k=1}^{r_h} \mathsf{dbf}(I_k^h)\right]\right) \\
&\{\text{by (1)}\} \\
\geq &\max\left(0, \Delta - \left[\sum_{k=1}^{r}(P(I_k) \cdot \Delta + R(I_k)) \right.\right. \\
&\left.\left. + \sum_{h=1}^{m}\sum_{k=1}^{r_h}(P(I_k^h) \cdot \Delta + R(I_k^h))\right]\right) \\
&\{\text{by Definition 8}\} \\
= &\max(0, \Delta - (F \cdot \Delta + G)) \\
&\{\text{by the definition of } \widehat{u_h} \text{ and } \sigma_h \\
&\text{in the statement of the lemma}\} \\
\geq &\max(0, \widehat{u_h} \cdot (\Delta - \sigma_h)).
\end{aligned}$$

By Definition 8 and (3), $\widehat{u_h}$ as defined in the statement of the lemma is positive. □

We next illustrate soft real-time schedulability analysis of a system with interrupts using Lemma 1 and Theorem 5.

Example 3. Consider the system from Example 2. The maximum tardiness for T_1's jobs may or may not be bounded depending on how interrupts are dispatched. We now analyze this task system using Theorem 5. By (2), the only global interrupt source I_1 considered has $\mathsf{dbf}(I_1, \Delta) = \left\lfloor \frac{\Delta}{1000} \right\rfloor \cdot 2 + \min\left(2,\ \Delta - \left\lfloor \frac{\Delta}{1000} \right\rfloor \cdot 2\right)$. The parameters $P(I_1)$ and $R(I_1)$ for which (1) holds are 0.002 and 2, respectively. Setting these parameters into Lemma 1, we find $\widehat{u_h} = 0.998$, and $\sigma_h = 2.004$ for $h = 1$ and 2. Applying Theorem 5 to this configuration, we find that (8) does not hold because $\sum_{h=1}^{m} \widehat{u_h} = 2 \cdot 0.998 < \max(2-1, 0) \cdot u_1 + u_1 = 2 \cdot 0.999$. Thus, bounded tardiness is not guaranteed for τ.

5 Special Cases

In the previous section, we assumed that all processors execute both ISRs and real-time jobs and made no assumptions about the type of interrupts. In this section, we discuss three special cases that are commonly encountered in real systems: periodic timer interrupts, dedicated interrupt handling, and timer multiplexing.

5.1 Timer Ticks under Task-Centric Accounting

Recall that, under task-centric accounting, the execution cost of a task $T_i(e_i, p_i)$ is inflated to account for all local ISR invocations on all processors that may occur during the maximum interval in which a single job of T_i can be active, *i.e.*, a job of T_i is charged for all delays in the interval in which it *might* execute. In the general case, this likely very pessimistic charge is unavoidable because a job may execute on all processors (due to migrations) and be delayed by all invoked ISRs on each processor (due to adverse timing of ISR invocations). In fact, such a scenario is trivial to construct assuming sporadic interrupt sources (in which case all ISR invocations may occur simultaneously) and sporadic job releases. However, a less-pessimistic bound can be derived for timer ticks (and similar interrupt sources) due to the regularity of their occurrence.

Definition 9. An interrupt is *periodic* if its minimum invocation separation equals its maximum invocation separation, and *replicated* if a corresponding ISR is invoked on *each* processor. Note

that ISRs are not necessarily invoked on each processor simultaneously (*i.e.*, "staggered" ISR invocations are allowed), but invocations on each processor must be periodic (*i.e.*, the "staggering" is constant).

Periodic ISR delay. Periodic interrupts differ from the more general model discussed in Section 3 in that we can assume an exact separation of ISR invocations, which enables us to apply techniques of classic response-time analysis (see (Liu, 2000) for an introduction). Let I_x denote a periodic, replicated interrupt source that, on each processor, triggers every p_x time units an ISR that executes for at most c_x time units.

Consider a job $T_{i,j}$ of task $T_i(e_i, p_i)$ such that e_i already accounts for all other sources of delays (*e.g.*, non-periodic global and local interrupts). If $T_{i,j}$ is not preempted while it executes, then, based on the response-time analysis of an equivalent two-task system under static-priority scheduling, $T_{i,j}$'s inflated execution cost due to I_x is given by the smallest e'_i that satisfies

$$e'_i = e_i + \left\lceil \frac{e'_i}{p_x} \right\rceil \cdot c_x. \tag{10}$$

Migrations. If $T_{i,j}$ migrates, then (10) may not hold in all cases—each time $T_{i,j}$ migrates, it might incur one additional ISR invocation because ISR invocations are not required to occur simultaneously on all processors. Similarly, if $T_{i,j}$ is preempted (but does not migrate), then it might still incur one additional ISR invocation when it resumes because its completion was delayed. Hence, if $T_{i,j}$ is preempted or migrates at most η times, then $T_{i,j}$'s inflated execution cost is given by the smallest e'_i that satisfies

$$e'_i = e_i + \left\lceil \frac{e'_i}{p_x} \right\rceil \cdot c_x + \eta \cdot c_x \tag{11}$$

Under G-EDF, jobs are only preempted by newly-released higher-priority jobs, *i.e.*, every job release causes at most one preemption to occur. Thus, η can be bounded based on the number of jobs with earlier deadlines that can be released while a job $T_{i,j}$ is pending using standard techniques (Liu, 2000).

5.2 Interrupt Handling on a Dedicated Processor

As discussed earlier in Section 4, both task-centric and processor-centric accounting methods can be pessimistic. For the task-centric method, we have assumed that each job of a task can be delayed by all (non-timer) ISRs invoked between its release time and deadline. For the processor-centric method, we have assumed that each processor is not available for scheduling real-time tasks during the execution of an ISR. We made these assumptions because, in general, it is neither known on exactly which processor a job executes, nor which processor handles an ISR corresponding to a global interrupt source.

Having more control over ISR execution may reduce ISR interference and improve the analysis. In this section, we consider a multiprocessor system in which all interrupts (except replicated timer ticks) are dispatched to a dedicated (master) processor that does not schedule real-time jobs (Stankovic and Ramamritham, 1991). (Without loss of generality we assume that this is processor 1.) The remaining $m - 1$ (slave) processors are used for running real-time tasks and only service IPI and timer-tick ISRs. The intuition behind this scheme is the following: by dispatching *most* interrupts in the system to the master processor, slave processors can service real-time jobs mostly undisturbed. The tradeoff is that one processor is "lost" to interrupt processing, with the benefit that pessimistic overhead accounting is (mostly) avoided on all slave processors. Thus, if accounting for all interrupt sources causes a utilization loss of greater than one due to pessimistic analysis, then dedicated interrupt processing should yield improved overall schedulability.

We assume that the ISRs on the master processor execute in FIFO order. When a job-release ISR completes on the master processor and some job running on a slave processor needs to be preempted, an IPI is dispatched to that processor. Thus, even though the majority of interrupt processing work has been moved to the master processor, the IPI service routines are still executed on slave processors.

Definition 10. In this section, we assume that the maximum cost of a job-release ISR is $c^{[I]}$.

Example 4. Consider a task set $\tau = \{T_1(1,4), T_2(1,4), T_3(2,12)\}$ running on a two-processor platform. Suppose that jobs of tasks in τ are released by ISRs executed on processor 1. Suppose

that the maximum job-release ISR cost and IPI service routine cost is $c^{[I]} = 0.5$ and rescheduling has zero cost. An example schedule for this system is shown in Figure 5. At time zero, job-release interrupts from the three sources I_1, I_2, and I_3 are dispatched to processor 1. An ISR for I_3 executes during the interval $[0, 0.5)$. At time 0.5, job $T_{3,1}$ is available for execution so an IPI is dispatched to processor 2. The IPI service routine executes on processor 2 during the interval $[0.5, 0.75)$ and job $T_{3,1}$ commences execution at time 0.75. Similarly, the releases of jobs $T_{2,1}$ and $T_{1,1}$ are processed.

In order to reduce the interference caused by IPIs, these interrupts are fired only if rescheduling needs to be done. In the schedule shown in Figure 5, the IPI corresponding to job $T_{2,2}$ is not sent at time 4.5 (when the respective job-release ISR completes) because job $T_{1,2}$ has higher priority than $T_{2,2}$ and thus does not need to be preempted. Since IPIs are only sent when rescheduling is required, their cost can be charged on per-job basis. As discussed in Section 4.1, we assume that said cost is included in each task's worst case execution time. Next, we need to account for the delay introduced by sequential interrupt processing on the master processor.

Definition 11. Let $\rho_{i,j}$ be the time when $T_{i,j}$ is available for scheduling. We call $\rho_{i,j}$ the *effective release time* of $T_{i,j}$.

Example 5. Consider jobs $T_{3,1}$, $T_{2,1}$, and $T_{1,1}$ from Example 4. Since the IPIs for these jobs are treated as job execution, these jobs have effective release times $\rho_{3,1} = 0.5$, $\rho_{2,1} = 1$, and $\rho_{1,1} = 1.5$, respectively, as shown in Figure 5. However, since the absolute deadlines for these jobs remain unchanged, the effective relative deadlines for these three jobs are 10.5, 3, and 2.5, respectively. For task T_1, the effective relative deadline is given by $D'_1 = D_1 - 3 \cdot c^{[I]} = p_1 - 3 \cdot c^{[I]} = 2.5$ as shown in Figure 5. Similarly, for task T_3, the effective relative deadline is given by $D'_3 = D_3 - 3 \cdot c^{[I]} = p_3 - 3 \cdot c^{[I]} = 10.5$ (see job $T_{3,2}$ in Figure 5).

Definition 12. Let $\max_{j \geq 1}(\rho_{i,j} - r_{i,j})$ be the maximum distance between the release time and effective release time of T_i's jobs. This distance depends on the total time the execution of a job-release ISR of T_i can be delayed. Let

$$J = \max \left(c^{[I]}, \sup_{\lambda \geq 0} \left(\sum_{x=1}^{r} \mathsf{dbf}(I_x, \lambda) + \sum_{x=1}^{r_1} \mathsf{dbf}(I_x^1, \lambda) - \lambda \right) \right).$$

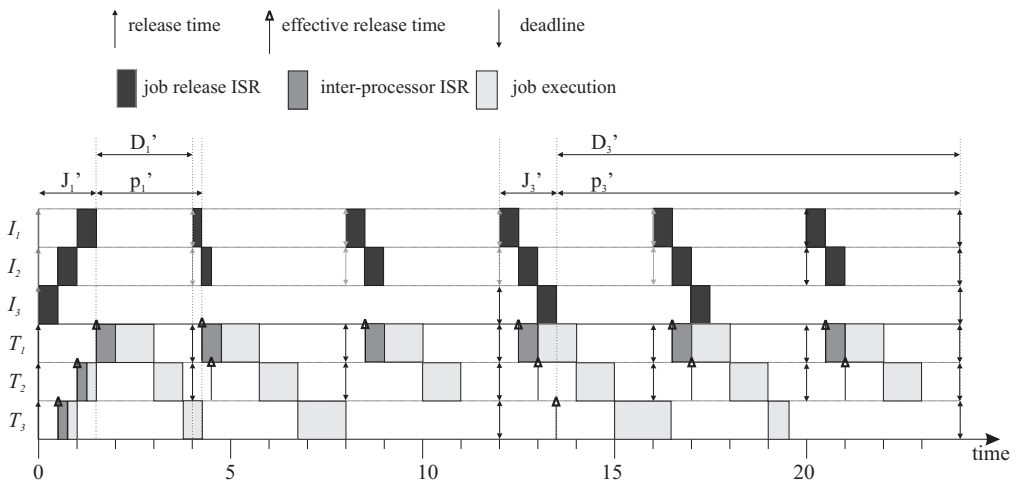

Figure 5: Interrupts processed on a dedicated processor (see Examples 4–5).

Lemma 2. $\max_{j \geq 1}(\rho_{i,j} - r_{i,j}) \leq J$.

Proof. The result of this lemma is similar to the formula for the maximum event delay from Network Calculus (LeBoudec and Thiran, 2001). However, our setting is slightly different due to the definition of service demand in Definition 1. Consider a job $T_{\ell,q}$ such that $\rho_{\ell,q} - r_{\ell,q} \geq c^{[I]}$. Let $c_{\ell,q}$ be the cost of $T_{\ell,q}$'s release ISR. Consider the latest time instant $t_0 \leq r_{\ell,q}$ such that no local or global ISR is pending on the master processor before t_0. $T_{\ell,q}$'s release ISR completes at or before all ISRs arriving within $[t_0, r_{\ell,q}]$ that are either global or local to processor 1 complete. From Definition 1, we have

$$\rho_{\ell,q} \leq t_0 + \sum_{x=1}^{r} \mathsf{dbf}(I_x, r_{\ell,q} - t_0) + \sum_{x=1}^{r_1} \mathsf{dbf}(I_x^1, r_{\ell,q} - t_0)$$

Subtracting $r_{\ell,q}$ from the inequality above, we have

$$\begin{aligned}
\rho_{\ell,q} - r_{\ell,q} &\leq t_0 - r_{\ell,q} + \sum_{x=1}^{r} \mathsf{dbf}(I_x, r_{\ell,q} - t_0) + \sum_{x=1}^{r_1} \mathsf{dbf}(I_x^1, r_{\ell,q} - t_0) \\
&\quad \{\text{setting } r_{\ell,q} - t_0 = \lambda\} \\
&= \sum_{x=1}^{r} \mathsf{dbf}(I_x, \lambda) + \sum_{x=1}^{r_1} \mathsf{dbf}(I_x^1, \lambda) - \lambda \\
&\quad \{\text{by the selection of } t_0 \text{ and } \lambda,\ \lambda \geq 0\} \\
&\leq \sup_{\lambda \geq 0} \left(\sum_{x=1}^{r} \mathsf{dbf}(I_x, \lambda) + \sum_{x=1}^{r_1} \mathsf{dbf}(I_x^1, \lambda) - \lambda \right). \qquad \square
\end{aligned}$$

The example below shows that sequential execution of job-release ISRs changes the minimum time between effective release times of subsequent jobs of a task.

Example 6. Consider jobs $T_{1,1}$ and $T_{2,1}$ in the previous example. Their effective release times are $\rho_{1,1} = 1.5$ and $\rho_{2,1} = 4.25$ as shown in Figure 5. As seen, the effective job releases of subsequent jobs of T_1 are separated by at least 2.5 time units. This distance is denoted with p_1'.

The theorem below presents a schedulability test. From an analysis standpoint, this test is a combination of the task- and processor-centric methods in the sense that both interrupt costs are charged on a per-task basis and that the available processor capacity is reduced.

Theorem 6. If $\tau' = \{T_1', \ldots, T_n'\}$, where $T_i'(e_i, p_i - J)$, is hard real-time schedulable (respectively, has bounded tardiness) on $m - 1$ processors, then τ is hard real-time schedulable (respectively, has bounded tardiness) on m-processor platform with all ISRs processed on a dedicated processor.

Proof. When interrupts are handled on a dedicated processor the minimum time between consecu-

tive effective job releases of T_i is

$$\begin{aligned}
\rho_{i,j+1} - \rho_{i,j} &= \rho_{i,j+1} - r_{i,j} + r_{i,j} - \rho_{i,j} \\
&\quad \{\text{by Lemma 2}\} \\
&\geq \rho_{i,j+1} - r_{i,j} - J \\
&\quad \{\text{because } \rho_{i,j+1} \geq r_{i,j+1}\} \\
&\geq r_{i,j+1} - r_{i,j} - J \\
&\quad \{\text{because } T_i \text{ is sporadic}\} \\
&\geq p_i - J.
\end{aligned}$$

If τ is not hard real-time schedulable or has unbounded tardiness, then an offending schedule for τ' can be constructed from an offending schedule for τ by mapping each job $T_{i,j}$ to the job $T'_{i,j}$ such that $\rho_{i,j} = r'_{i,j}$ and $d_{i,j} = d'_{i,j}$. □

Note that, in contrast to Theorem 1, we have subtracted the interrupt-induced delay J from the task's period. This is beneficial because $\frac{e}{p-J} \leq \frac{e+J}{p}$ for all positive p and non-negative e and J such that $p \geq e + J$. If $p < e + J$, then the task set is claimed unschedulable.

Finally, if slave processors service timer ISRs, then the schedulability of the transformed task set τ' on $m - 1$ processors should be checked after applying the task-centric analysis of timer ticks (Section 5.1).

5.3 Timer Multiplexing

Up to this point, we have made no assumptions regarding the type of ISRs that cause jobs to be released, *i.e.*, the above analysis applies to any system in which job releases are triggered by either DI and TI sources (recall Section 2.1).

An important special case are systems in which *all* job releases are time-triggered. For example, in multimedia systems, video processing is usually frame-based, where each frame corresponds to a point in time. Similarly, process control applications typically require each controller to be

activated periodically (Liu, 2000). Further, if devices are operated in polling mode, then each job that polls a device is invoked in response to an expiring timer.

Since the number of programmable timer devices is typically small (*e.g.*, there is only one (processor-local) timer guaranteed to be available in Sun's sun4v architecture (Saulsbury, 2008)), operating systems are commonly designed such that a (possibly large) number of *software timers* is multiplexed onto a single (or few) hardware timer(s) (Varghese and Lauck, 1987). For example, in recent versions of Linux, this is accomplished by programming a one-shot hardware timer to generate an interrupt at the earliest future expiration time and then processing all expired software timers during each timer ISR invocation (Gleixner and Niehaus, 2006). If timers expire while an ISR is being serviced, then such expirations are handled before re-programming the hardware timer. Hence, job releases that occur "almost together" only require a single interrupt to be serviced.

If all job releases are triggered by software timers and if all interrupts are serviced by a dedicated processor (as discussed in Section 5.2 above) then such multiplexing of a hardware timer can be exploited for better analysis as follows. As before, let $c^{[I]}$ denote the maximum cost of a job-release ISR, *i.e.*, the maximum time required to service a single one-shot timer interrupt, and let J denote the maximum release delay due to ISR processing as in Lemma 2.

According to Definition 12 and Lemma 2, in the absence of interrupt sources other than those corresponding to the job releases of each task, $J \geq n \cdot c^{[I]}$, *i.e.*, if there are up to $n - 1$ earlier ISRs of length $c^{[I]}$, then the effective release time of a job can be delayed by at least $n \cdot c^{[I]}$ time units. This is a lower bound in the general case, wherein each task might correspond to a different DI source.

However, this is not the case under timer multiplexing: if all job releases are triggered by (software) timers, then coinciding job releases are handled by the same ISR and thus $J = c^{[I]}$. If n is large, then this observation can amount to a substantially reduced bound on worst-case release jitter.

6 Experimental Evaluation

To assess the effectiveness of the interrupt accounting approaches described above, we conducted an experimental evaluation in which the methods were compared based on how many randomly-generated task sets could be claimed schedulable (both hard and soft) if interrupts are accounted for using each method.

6.1 Experimental Setup

Similarly to the experiments previously performed in (Brandenburg et al., 2008; Brandenburg and Anderson, 2009), we used distributions proposed by Baker (2005) to generate task sets randomly. Task periods were uniformly distributed over [10ms, 100ms]. Task utilizations were distributed differently for each experiment using three uniform and three bimodal distributions. The ranges for the uniform distributions were [0.001, 0.1] (*light*), [0.1, 0.4] (*medium*), and [0.5, 0.9] (*heavy*), respectively. In the three bimodal distributions, utilizations were distributed uniformly over either [0.001, 0.5) or [0.5, 0.9] with respective probabilities of 8/9 and 1/9 (*light*), 6/9 and 3/9 (*medium*), and 4/9 and 5/9 (*heavy*).

We considered ISRs that were previously measured on a Sun Niagara multicore processor in (Brandenburg and Anderson, 2009); namely, the one-shot timer ISR (global, unless handled by dedicated processor), the periodic, replicated timer tick ISR signaling the beginning of a new quantum (local; $Q = 1000\mu$s), and the IPI delay (see Section 4.2). The Niagara is a 64-bit machine containing eight cores on one chip clocked at 1.2 GHz. Each core supports four hardware threads, for a total of 32 logical processors. These hardware threads are real-time-friendly because each core distributes cycles in a round-robin manner—in the worst case, a hardware thread can utilize every fourth cycle. On-chip caches include a 16K (resp., 8K) four-way set associative L1 instruction (resp., data) cache per core, and a shared, unified 3 MB 12-way set associative L2 cache. Our test system is configured with 16 GB of off-chip main memory.

We assumed worst-case (resp., average-case) ISR costs as given in (Brandenburg and Anderson, 2009) and reproduced in Table 1 when testing hard (resp., soft) schedulability. As we are only

	Average Case			Worst Case		
n	**Release**	**Timer Tick**	**IPI delay**	**Release**	**Timer Tick**	**IPI delay**
50	13.49	1.73	3.62	45.38	8.88	6.55
100	20.40	1.78	3.37	88.88	9.23	5.00
150	28.43	1.84	3.31	135.91	9.38	5.16
200	36.11	1.89	3.29	150.63	9.57	5.02
250	44.96	1.92	3.30	164.73	9.61	5.08
300	51.64	1.95	3.33	174.49	9.71	7.15
350	60.85	1.98	3.35	186.64	9.85	5.40
400	69.75	2.01	3.40	200.14	9.90	9.43
450	78.90	2.04	3.49	217.46	10.03	7.52

Table 1: Worst-case and average-case ISR invocation costs under G-EDF (in μs) as a function of the task set size n (from (Brandenburg and Anderson, 2009)). We used non-decreasing piece-wise linear interpolation to derive ISR costs for values of n between measured task set sizes.

concerned with interrupt accounting, all other sources of overhead (scheduling, cache-related, *etc.*) were considered negligible.[7]

To assess the impact (or lack thereof) of possible improvements in the OS implementation, we tested each method twice: once assuming full ISR costs as given in (Brandenburg and Anderson, 2009), and once assuming ISR costs had been reduced by 80%. We consider the latter to be an approximate upper bound on the best-case scenario based on the conjecture that overheads likely cannot be reduced by *more* than 80%.

For each task set, we applied each of the accounting methods described in Section 4, and also assuming dedicated interrupt handling (Section 5.2) and timer multiplexing (Section 5.3). To determine whether an inflated task system τ' is hard real-time schedulable under the quantum-centric and task-centric methods and dedicated interrupt handling, we used all major published sufficient (but not necessary) hard real-time schedulability tests for G-EDF (Goossens et al., 2003; Baker,

[7] An implementation study considering full overheads and other implementation choices beyond the scope of this paper has recently been completed (Brandenburg and Anderson, 2009).

2003; Bertogna et al., 2005; Baruah, 2007; Bertogna et al., 2009) and deemed τ' to be schedulable if it passes at least one of these five tests. We assumed a quantum size of 1 millisecond, which is a common choice in Linux-based systems such as the one considered in (Brandenburg and Anderson, 2009).

To determine hard real-time schedulability under the processor-centric method, we used Shin et al.'s *Multiprocessor Resource* (MPR) test (Shin et al., 2008) and chose $\max\{p_i\}$ as the MPR period for analysis purposes. As an optimization, we used the supply-bound function given in (12) below, which is less pessimistic than the (more general) one given by Shin et al. (2008).

$$\mathrm{sbf}(\Delta) = m\left(\Delta - \sum_{h=1}^{m}\sum_{k=1}^{r_h} \mathsf{dbf}(I_k^h, \Delta) - \sum_{k=1}^{r} \mathsf{dbf}(I_k, \Delta)\right) \tag{12}$$

Note that in (Shin et al., 2008) the supply-bound function is derived based on a given period and supply. In contrast, we compute the supply for a chosen period with (12) (for $\Delta = \max\{p_i\}$).

Recall from Section 4.1 that the scheduling model underlying quantum-centric analysis is different from the event-driven model assumed by task- and processor-centric analysis. Thus, schedulability under quantum-centric accounting is affected not only by interrupt delays, but also by utilization loss due to the quantization of task parameters. This is reflected in our results since it is fundamental to quantum-driven scheduling and hence an inherent source of pessimism that should be considered when comparing analysis methods.

6.2 Results

Our schedulability results are shown in Figures 6–17: Figures 6–11 depict hard real-time results, and Figures 12–17 depict soft real-time results. Inset (a) of each figure shows schedulability assuming full ISR costs, whereas inset (b) of each figure shows schedulability of the same collection of task sets assuming ISR costs reduced by 80%. Sampling points were chosen such that the sampling density is high in areas where curves change rapidly. For each sampling point, we tested 1,000 task sets, for a total of over 2,100,000 task sets.

6.2.1 Hard Real-Time Results Assuming Full ISR Costs

Figures 6–11 show the ratio of task sets claimed hard real-time schedulable by the five methods. In all scenarios, the quantum-centric method shows consistently disappointing performance. In Figure 6(a), which corresponds to the motivating example in Section 1, even with utilization less than two, most task sets were deemed unschedulable using this method. Performance improves with heavier task sets (*e.g.*, see Figure 10(a)), but never becomes competitive due to the overall high number of tasks.

The processor-centric method works best if there are no heavy tasks (see Figure 6(a)), but does not perform better than the task-centric accounting method in any of the considered scenarios—unfortunately, the pessimism in the processor-centric method's adaptation of Shin et al. (2008)'s MPR test exceeds the pessimism of the task-centric method. In the case of bimodal utilization distributions (inset (a) of Figures 7, 9, and 11), the processor-centric method does not perform significantly better than quantum-centric accounting due to the presence of heavy tasks in almost all task sets. Similar trends are apparent for uniform distributions in inset (a) of Figures 8 and 10.

In Figure 6(a), the task-centric accounting method is pessimistic due to large inflation costs in accordance with Theorem 4, but as the number of tasks decreases (the average per-task utilization increases), its performance improves significantly, as can be seen in inset (a) of Figures 7–11.

Dedicated interrupt handling is preferable to task-centric accounting if task counts are high, and particularly so if timer multiplexing is assumed. For example, in Figure 6(a), the combination of a dedicated processor and timer multiplexing can support loads five times larger than those supported by task-centric accounting. The differences are less pronounced under bimodal distributions (insets (a) of Figures 7 and 9), which generally feature lower task counts than their uniform counter parts. Also note that timer multiplexing does not yield improved schedulability compared to regular dedicated interrupt processing in these cases.

Dedicated interrupt handling is not preferable under both uniform and bimodal heavy distributions, as can be seen in inset (a) of Figures 10 and 11. In these cases, the utilization lost due to dedicating a processor is not outweighed by the benefits of avoiding interrupts on the processors on which real-time jobs execute.

6.2.2 Soft Real-Time Results Assuming Full ISR Costs

Figures 12–17 show soft real-time schedulability results. In a reversal of the hard real-time trends, for the uniform light and medium utilization ranges, the performance of the processor-centric method is superior to the performance of the task-centric method (see inset (a) of Figures 12 and 14). This is because (7) and (8) in Theorem 5 are likely to hold in the absence of heavy tasks. In contrast, the processor-centric method is exceptionally pessimistic in the case of the uniform heavy distribution (see Figure 16(a)). This is because we assumed that all processors are not available for the duration of an ISR, which leads to a violation of (8) if heavy tasks are present, due to the term $\max(H-1,0)\cdot\max(u_i)$ (note that $H=m$). For the same reason, task sets are pessimistically claimed unschedulable under the bimodal utilization distributions (inset (a) of Figures 13, 15, and 17). In fact, the processor-centric method offers (almost) no advantage over the quantum-centric method in the soft real-time case if heavy tasks are present at all. This is in stark contrast to the task-centric method, which is consistently the best of the three general interrupt accounting methods unless task counts are high (inset (a) of Figures 13, 15, 16, and 17).

Dedicated interrupt handling is preferable to task-centric accounting for all considered utilization distributions. However, in the cases in which processor-centric accounting performs well (inset (a) of Figures 12 and 14), processor-centric accounting is preferable to regular dedicated interrupt handling. This is not the case for dedicated interrupt processing in combination with timer multiplexing, which always yields the best soft real-time schedulability.

6.2.3 Results Assuming Reduced ISR Costs

The experiments assuming reduced release overhead reveals six major trends:

1. under the considered bimodal distributions, hard real-time schedulability of the best-performing methods does not significantly improve with reduced ISR costs (Figures 7, 9, 11)—this is likely indicative of fundamental limitations of the G-EDF schedulability tests published to date (or, possibly, G-EDF itself) with regard to scheduling mixed-utilization task sets;

2. reducing overhead helps only little to overcome the task-centric method's asymptotic growth

(Figures 6 and 12);

3. even with reduced overhead, the quantum-centric and processor-centric methods are not competitive in the hard real-time case;

4. in the soft real-time case, the processor-centric method performs equally badly assuming either 100% or 20% overhead if heavy tasks are present (see Figures 13, 15, 16, and 17);

5. the quantum-centric method's competitiveness is much improved in the soft real-time case if overhead is reduced; and

6. in the soft real-time case, dedicated interrupt handling in combination with timer multiplexing is almost always the best-performing method, with only Figure 14(b) as a notable exception.

These results show that the choice of interrupt accounting method can have significant impact on schedulability. Further, it appears worthwhile to improve both OS implementations (to reduce overheads) and analysis techniques (to reduce inherent pessimism). Especially promising is the performance of the processor-centric accounting method in the soft real-time case in the absence of heavy tasks—further research is warranted to identify and remove the limitations that impede processor-centric accounting in the presence of heavy tasks.

Limitations. Because the focus of this paper is primarily the analysis of interrupt accounting methods (and since this is not an implementation study), we have made the simplifying assumption that ISR costs are equal under both quantum-driven scheduling, event-driven scheduling, and dedicated interrupt handling. Thus, results may deviate somewhat in practice depending on hardware and OS characteristics; however, the fundamental tradeoffs and limitations exposed by our experiments remain valid.

7 Conclusion

This paper surveyed various types of interrupts that occur in multiprocessor real-time systems and summarized Devi's quantum-based accounting method. Two recently-developed approaches to

accounting for interrupt-related delays under G-EDF scheduling for both hard and soft real-time systems were presented: the task-centric method and the processor-centric method. Further, analysis for the common practice of dedicating a processor to interrupts to shield real-time jobs from delays was presented, including the special case of timer multiplexing.

In an empirical comparison, the task-centric method performed well in most of the tested scenarios; however, it is subject to utilization loss that is quadratic in the number of tasks. Hence, it has inferior performance for large task sets. In contrast, in the soft real-time case, the processor-centric method performed well for task systems with many light tasks, but yielded overly pessimistic results in the presence of heavy tasks. In all considered scenarios, at least one of the two new methods performed significantly better than the previously-proposed quantum-centric method.

Further, the experiments revealed that dedicating a processor to interrupt handling can be beneficial to schedulability in many cases—especially so if all job releases are time-triggered. However, dedicated interrupt handling is likely much less competitive for lower processor counts, and is also not always the best choice if overheads are low. Thus, accounting for interrupts remains an important component of schedulability analysis.

In future work, we would like to refine the processor-centric method to be less pessimistic with regard to both maximum task utilization and reductions in supply. Further, it may be beneficial to investigate whether multiprocessor real-time calculus (Leontyev et al., 2009) can be applied to account for interrupt delays.

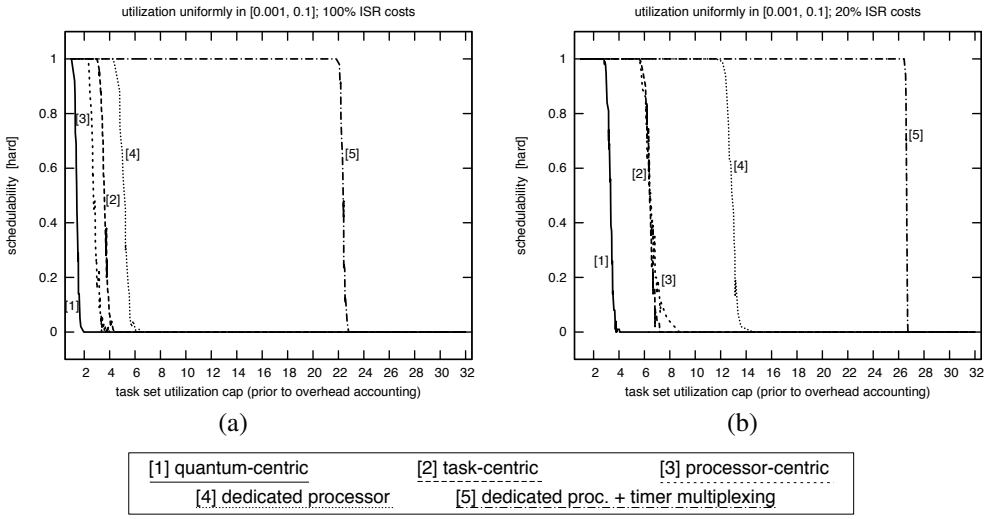

Figure 6: Hard schedulability (the ratio of task sets deemed schedulable) results for the uniform light utilization distribution. **(a)** Full ISR costs; **(b)** reduced ISR costs.

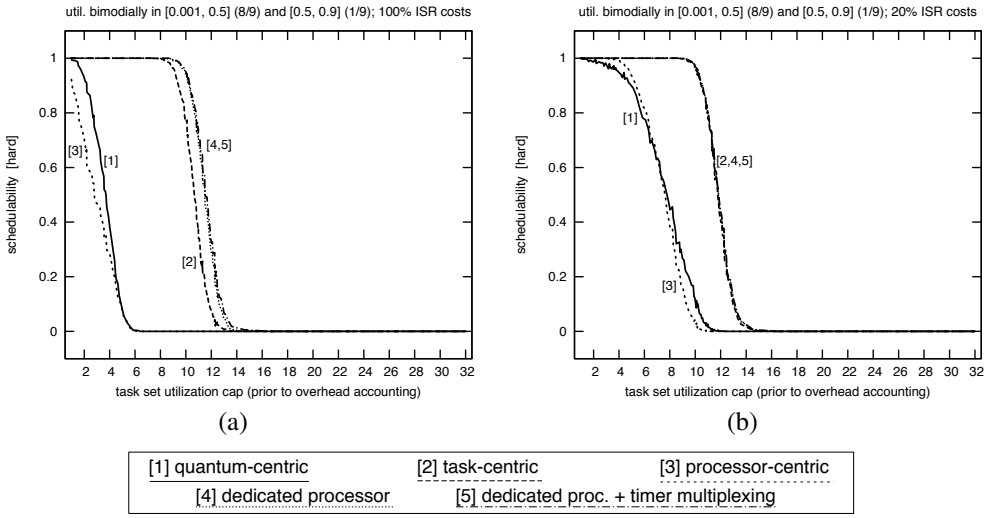

Figure 7: Hard schedulability (the ratio of task sets deemed schedulable) results for the bimodal light utilization distribution. **(a)** Full ISR costs; **(b)** reduced ISR costs.

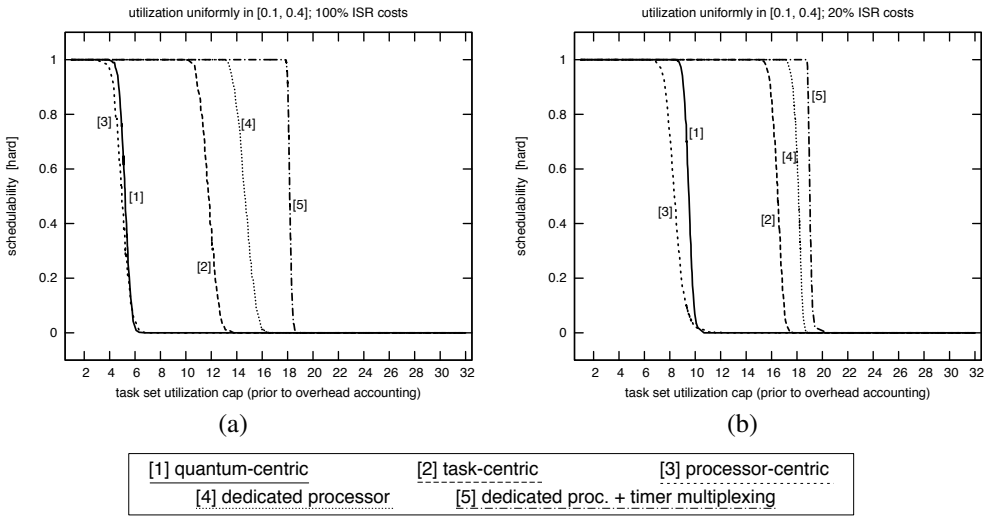

Figure 8: Hard schedulability (the ratio of task sets deemed schedulable) results for the uniform medium utilization distribution. **(a)** Full ISR costs; **(b)** reduced ISR costs.

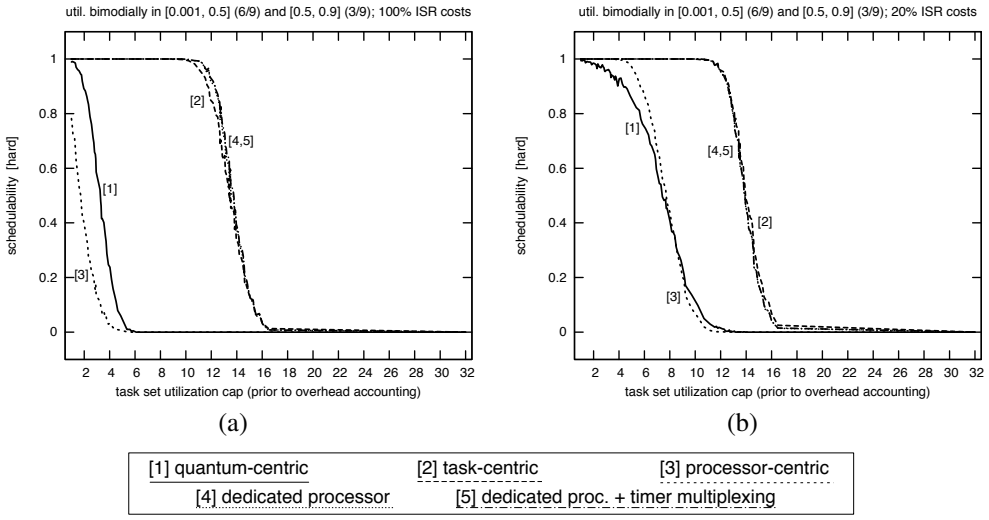

Figure 9: Hard schedulability (the ratio of task sets deemed schedulable) results for the bimodal medium utilization distribution. **(a)** Full ISR costs; **(b)** reduced ISR costs.

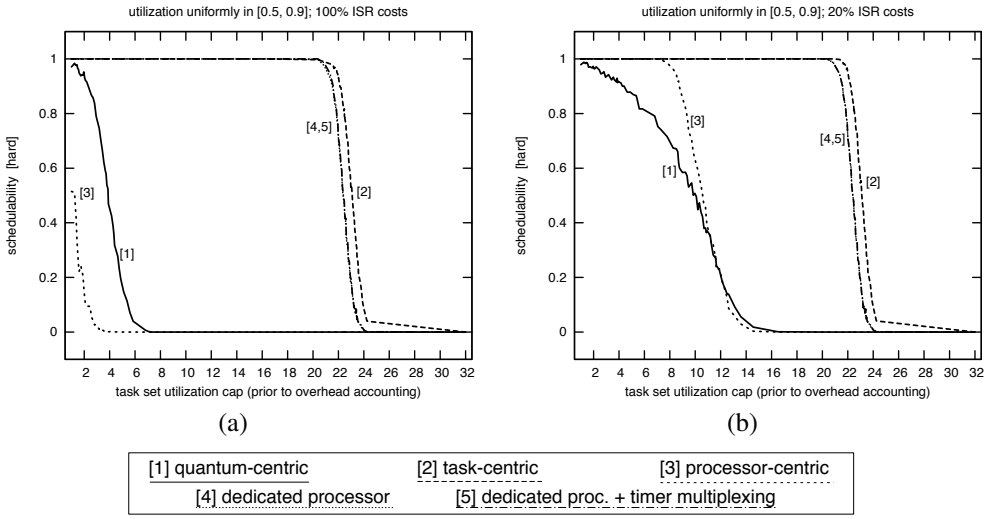

Figure 10: Hard schedulability (the ratio of task sets deemed schedulable) results for the uniform heavy utilization distribution. **(a)** Full ISR costs; **(b)** reduced ISR costs.

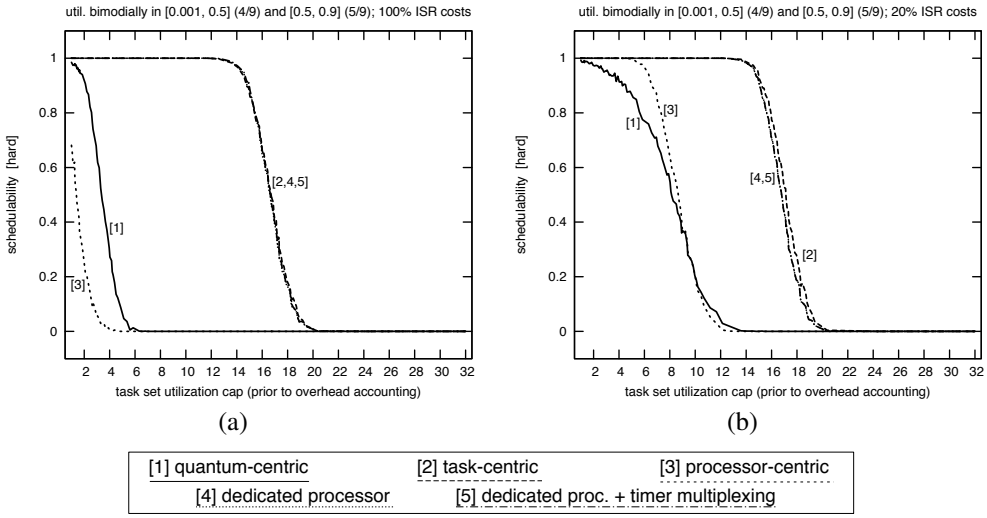

Figure 11: Hard schedulability (the ratio of task sets deemed schedulable) results for the bimodal heavy utilization distribution. **(a)** Full ISR costs; **(b)** reduced ISR costs.

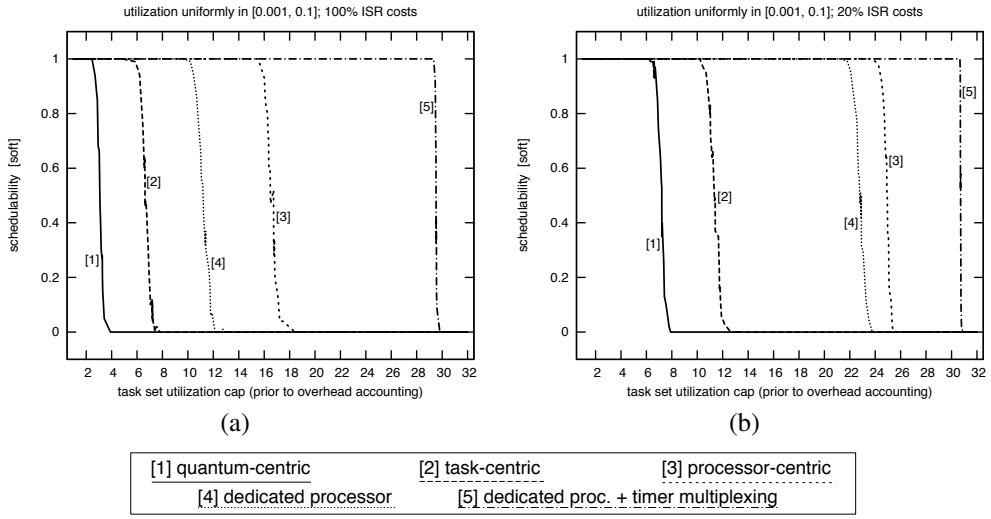

Figure 12: Soft schedulability (the ratio of task sets deemed schedulable) results for the uniform light utilization distribution. **(a)** Full ISR costs; **(b)** reduced ISR costs.

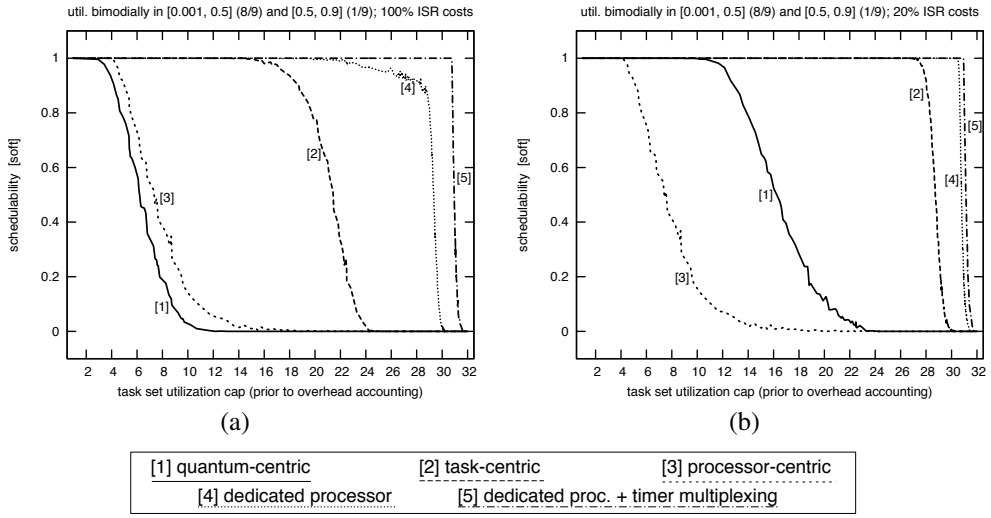

Figure 13: Soft schedulability (the ratio of task sets deemed schedulable) results for the bimodal light utilization distribution. **(a)** Full ISR costs; **(b)** reduced ISR costs.

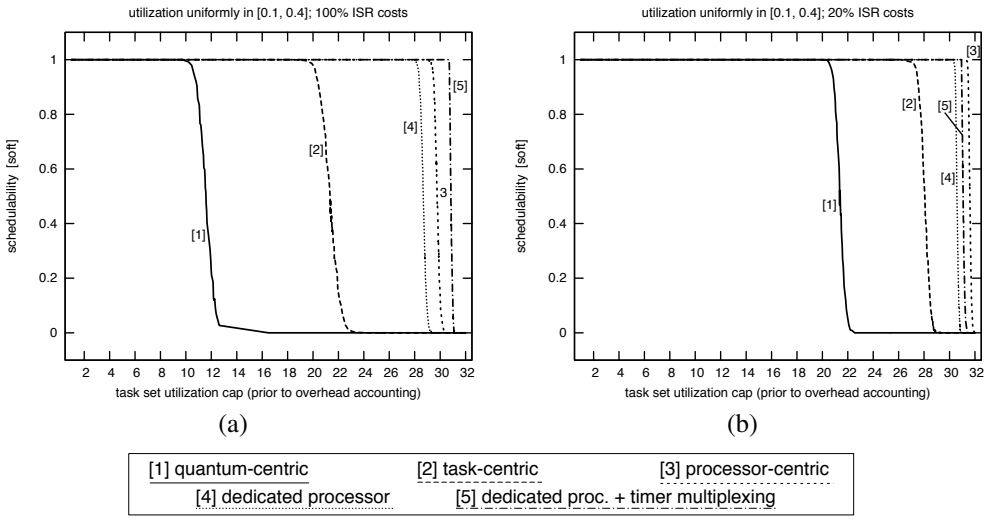

Figure 14: Soft schedulability (the ratio of task sets deemed schedulable) results for the uniform medium utilization distribution. **(a)** Full ISR costs; **(b)** reduced ISR costs.

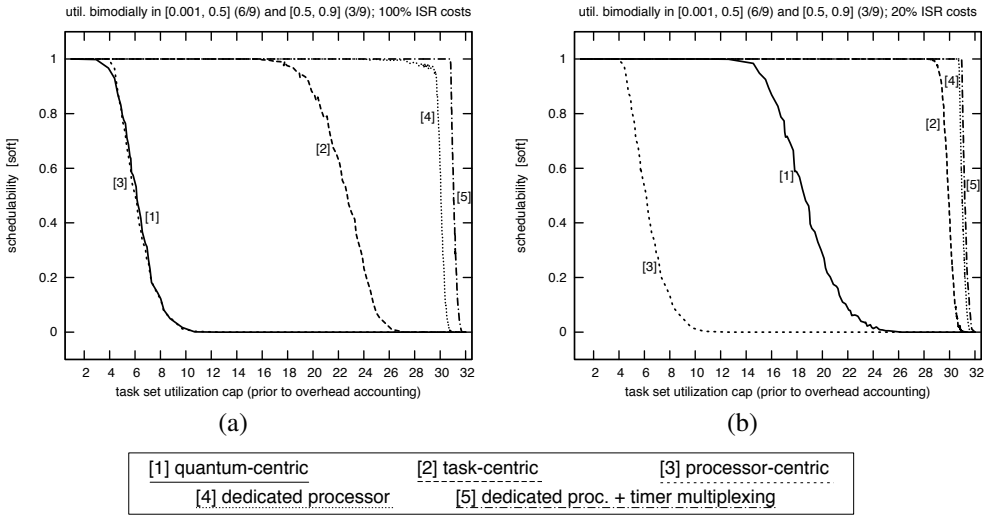

Figure 15: Soft schedulability (the ratio of task sets deemed schedulable) results for the bimodal medium utilization distribution. **(a)** Full ISR costs; **(b)** reduced ISR costs.

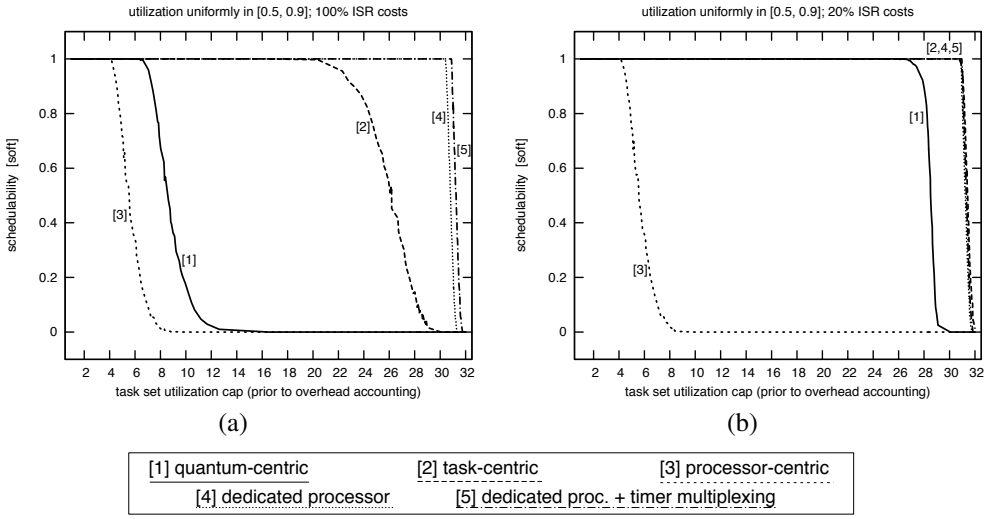

Figure 16: Soft schedulability (the ratio of task sets deemed schedulable) results for the uniform heavy utilization distribution. **(a)** Full ISR costs; **(b)** reduced ISR costs.

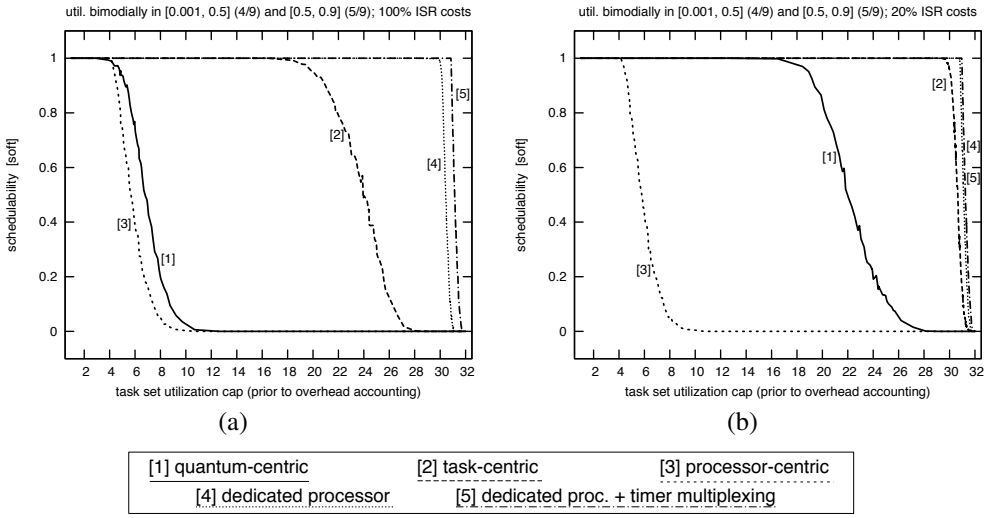

Figure 17: Soft schedulability (the ratio of task sets deemed schedulable) results for the bimodal heavy utilization distribution. **(a)** Full ISR costs; **(b)** reduced ISR costs.

References

D. Cofer, M. Rangarajan, Formal Verification of Overhead Accounting in an Avionics RTOS, in: Proceedings of the 23rd IEEE Real-Time Systems Symposium, 181–190, 2002.

J. Liu, Real-Time Systems, Prentice Hall, 2000.

C. Liu, J. Layland, Scheduling Algorithms for multiprogramming in a Hard Real-Time Environment, Journal of the ACM 30 (1973) 46–61.

K. Jeffay, D. Stone, Accounting for interrupt handling costs in dynamic priority task systems, Proceedings of the 14th Real-Time Systems Symposium (1993) 212–221.

U. Devi, Soft Real-Time Scheduling on Multiprocessors, Ph.D. thesis, University of North Carolina, Chapel Hill, North Carolina, 2006.

B. Brandenburg, J. Calandrino, J. Anderson, On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study, in: Proceedings of the 29th IEEE Real-Time Systems Symposium, 157–169, 2008.

B. Brandenburg, J. Anderson, A Comparison of the M-PCP, D-PCP, and FMLP on LITMUS$^{\mathrm{RT}}$, in: Proceedings of the 12th International Conference On Principles Of Distributed Systems, 105–124, 2008.

B. Brandenburg, J. Anderson, On the Implementation of Global Real-Time Schedulers, in: Proceedings of the 30th IEEE Real-Time Systems Symposium, 214–224, 2009.

B. Brandenburg, H. Leontyev, 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, 273–283, 2009.

Intel Corp., Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture, Intel Corp., 2008a.

Intel Corp., Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3: System Programming Guide, Intel Corp., 2008b.

D. Weaver, T. Germond (Eds.), The SPARC Architecture Manual. Version 9, PTR Prentice Hall, 1994.

A. Saulsbury, UltraSPARC Virtual Machine Specification, SUN Corp., 2008.

J. Kurzak, A. Buttari, P. Luszczek, J. Dongarra, The PlayStation 3 for High-Performance Scientific Computing, Computing in Science & Engineering 10 (3) (2008) 84–87.

Intel Corp., TLBs, Paging-Structure Caches, and Their Invalidation, Intel Corp., 2008c.

J. Stankovic, K. Ramamritham, The Spring Kernel: A New Paradigm for Real-Time Systems, IEEE Software 8 (3) (1991) 62–72.

S. Brosky, S. Rotolo, Shielded Processors: Guaranteeing Sub-millisecond Response in Standard Linux, in: Proceedings of the 17th International Symposium on Parallel and Distributed Processing (CD-ROM/Abstracts), 120.1, 2003.

E. Piel, P. Marquet, J. Soula, J.-L. Dekeyser, Asymmetric Scheduling and Load Balancing for Real-Time on Linux SMP, in: Proceedings of the Workshop on Scheduling for Parallel Computing (LNCS 3911), Springer Verlag, 869–903, 2005.

E. Betti, D. P. Bovet, M. Cesati, R. Gioiosa, Hard Real-Time Performances in Multiprocessor-Embedded Systems using ASMP-Linux, EURASIP Journal of Embedded Systems 2008 (2008) 1–16.

T. Baker, S. Baruah, Schedulability Analysis of Multiprocessor Sporadic Task Systems, in: S. H. Son, I. Lee, J. Y. Leung (Eds.), Handbook of Real-Time and Embedded Systems, Chapman Hall/CRC, Boca Raton, Florida, 2007.

J. Calandrino, J. Anderson, D. Baumberger, A Hybrid Real-Time Scheduling Approach for Large-Scale Multicore Platforms, in: Proceedings of the 19th Euromicro Conference on Real-Time Systems, 247–256, 2007.

J. Goossens, S. Funk, S. Baruah, Priority-driven scheduling of periodic task systems on multiprocessors, Real-Time Systems 25 (2-3) (2003) 187–205.

T. Baker, Multiprocessor EDF and deadline monotonic schedulability analysis, in: Proceedings of the 24th IEEE Real-Time Systems Symposium, 120–129, 2003.

M. Bertogna, M. Cirinei, G. Lipari, Improved Schedulability Analysis of EDF on Multiprocessor Platforms, in: Proceedings of the 17th Euromicro Conference on Real-Time Systems, 209–218, 2005.

S. Baruah, Techniques for multiprocessor global schedulability analysis, in: Proceedings of the 28th IEEE Real-Time Systems Symposium, 119–128, 2007.

H. Leontyev, J. Anderson, A Unified Hard/Soft Real-Time Schedulability Test for Global EDF Multiprocessor Scheduling, in: Proceedings of the 29th IEEE Real-Time Systems Symposium, 375–384, 2008.

M. Bertogna, M. Cirinei, G. Lipari, Schedulability analysis of global scheduling algorithms on multiprocessor platforms, IEEE Transactions on Parallel and Distributed Systems 20 (4) (2009) 553–566.

S. Baruah, A. Burns, Sustainable Scheduling Analysis, in: Proceedings of the 27th IEEE Real-Time Systems Symposium, 159–168, 2006.

H. Leontyev, J. Anderson, Generalized Tardiness Bounds for Global Multiprocessor Scheduling, in: Proceedings of the 28th IEEE Real-Time Systems Symposium, 413–422, 2007.

I. Shin, A. Easwaran, I. Lee, Hierarchical Scheduling Framework for Virtual Clustering of Multiprocessors, in: Proceedings of the 20th Euromicro Conference on Real-Time Systems, 181–190, 2008.

R. Ha, J. Liu, Validating timing constraints in multiprocessor and distributed real-Time systems, in: Proceedings of the 14th IEEE International Conference on Distributed Computing Systems, 162–171, 1994.

H. Leontyev, S. Chakraborty, J. Anderson, Multiprocessor Extensions to Real-Time Calculus, in: Proceedings of the 30th IEEE Real-Time Systems Symposium, to appear, 2009.

S. Chakraborty, Y. Liu, N. Stoimenov, L. Thiele, E. Wandeler, Interface-based Rate Analysis of Embedded Systems, in: Proceedings of the 27th IEEE Real-Time Systems Symposium, 25–34, 2006.

J. Y. LeBoudec, P. Thiran, Network Calculus - A Theory of Deterministic Queuing Systems for the Internet., 2050, Springer Verlag, 2001.

G. Varghese, T. Lauck, Hashed and hierarchical timing wheels: data structures for the efficient implementation of a timer facility, SIGOPS Operating Systems Review 21 (5) (1987) 25–38.

T. Gleixner, D. Niehaus, hrtimers and beyond: transformation of the Linux time(r) system, in: Proceedings of the 8th Linux Symposium, 333–346, 2006.

T. Baker, A Comparison of Global and Partitioned EDF Schedulability Tests for Multiprocessors, Tech. Rep. TR-051101, Florida State University, 2005.