Integrating Hard/Soft Real-Time Tasks and Best-Effort Jobs on Multiprocessors*

Björn B. Brandenburg and James H. Anderson
The University of North Carolina at Chapel Hill

Abstract

We present a multiprocessor scheduling framework for integrating hard and soft real-time tasks and best-effort jobs. This framework allows for full system utilization, and ensures that hard real-time deadlines are met and that deadline tardiness is bounded for soft real-time tasks. Dynamic slack reclamation is employed to reduce tardiness and to improve the response time of best-effort jobs. The approach is validated using an implementation within the Linux kernel.

1 Introduction

An important trend in computing is the ongoing move towards system- and chip-level parallelism. Because of heat and power issues, it has become increasingly difficult to improve processor performance by increasing clock speeds. Therefore, in order to continue performance improvements, major processor manufacturers, such as Intel, AMD, IBM, and Sun Microsystems, have embraced *multicore architectures*, which combine several processing cores on a single chip. It is expected that most future commodity chips will consist of multiple cores.

This development coincides with the emergence of applications for which both timing correctness and high performance are required. Examples include multimedia and entertainment applications and some business computing applications that require guaranteed transaction response times [14]. Such real-time applications have performance demands that, in the foreseeable future, can only be satisfied by inherently parallel systems. Vendors are responding to this need. For example, IBM's Cell processor, which includes a PowerPC and eight specialized cores on the same chip, was originally designed with gaming applications in mind. As another example, Azul Systems has developed the Vega2 system, which is a Java-based appliance with up to 768 cores for processing time-sensitive business transactions [4]. In the embedded systems arena, increasing processing-capacity demands and tight power-consumption constraints are also making the use of multicore systems increasingly widespread [3].

The problem. These emerging real-time workloads have highly heterogeneous timing requirements and may consist of hard real-time (HRT), soft real-time (SRT), and best-effort (BE) components. However, any HRT component, if present, will likely be small in terms of system utilization [16]. To enable such systems to be developed, a framework is needed for integrating SRT tasks and BE aperiodic jobs[1] with a relatively small number of HRT tasks such that **(i)** HRT deadlines can be guaranteed, **(ii)** deadline tardiness for SRT tasks and response times for BE jobs can be minimized to the extent possible, and **(iii)** the multiprocessor platform is efficiently utilized. In this paper, we consider the problem of devising and implementing such a framework.

Background. As in other work on integrating workloads with different requirements, we use a server-based abstraction. Such abstractions were first considered in the context of uniprocessor systems.[2] The general idea here is to partition the tasks to be scheduled among various "server tasks." A two-level scheduling approach is then used, where at the top level, server tasks are scheduled, and at the next level, the servers themselves schedule their constituent tasks. In this way, tasks with different timing requirements can be isolated from one another and dealt with independently.

A common method for real-time scheduling on multiprocessor systems is to partition the tasks to be scheduled among the available processors. While simple to implement, this approach has several drawbacks when implementing servers. For example, due to bin-packing issues, large parts of a system's capacity may have to remain unallocated. Further, partitioning is inflexible in that spare capacities cannot be reused on different processors. For these reasons, prior efforts on implementing servers on multiprocessors have focused on the usage of global scheduling algorithms (which allow tasks to migrate) as the top-level scheduler. Two different types of global algorithms have been considered: job-level fixed-priority algorithms and Pfair algorithms. In a *job-level fixed-priority algorithm*, a job's priority, once assigned, does not change. A notable example of such an algorithm is global EDF. In contrast, *Pfair algorithms* break each job to be scheduled into quantum-length pieces of work, called *subtasks*, which are then scheduled. Pfair algorithms, in theory, have the advantage that a system's full capacity can be

*Work supported by a grant from Intel Corp., by NSF grants CNS 0408996, CCF 0541056, and CNS 0615197 and by ARO grant W911NF-06-1-0425. The first author was also supported by a Fulbright fellowship.

[1] A *task* is invoked repeatedly, with each invocation called a *job*. An *aperiodic job* is invoked once.

[2] The literature on uniprocessor server approaches is quite extensive. Due to space constraints, we focus most of our attention on the multiprocessor case. However, some of the ideas utilized in this paper first appeared in work on uniprocessors. We review this prior work in later sections where it is utilized.

utilized: any sporadic real-time task system with total utilization at most m can be scheduled using Pfair algorithms with no deadline misses on an m-processor system [18]. In contrast, as with partitioning, caps on overall utilization must be used when using job-level fixed-priority algorithms if every deadline must be met.

Related work. The problem of implementing servers on a multiprocessor was first considered by Baruah and Lipari [2], who presented a multiprocessor implementation of the uniprocessor total bandwidth server [17]. The problem addressed in this work is that of integrating on a multiprocessor the processing of aperiodic jobs, which are served by a total-bandwidth server, with the processing of jobs of recurrent HRT tasks. The paper focuses on using global EDF as the top-level scheduler, but the authors note that their results are applicable if other job-level fixed-priority algorithms or Pfair algorithms are used. In later work, Baruah *et al.* [1] presented a scheme called M-CBS, which extends the uniprocessor constant-bandwidth server scheme [5] for application on multiprocessor platforms. Each such server can encapsulate a collection of tasks with different requirements (perhaps entire applications). In this work, a variant of global EDF is used as the top-level scheduler that gives higher priority to certain high-utilization servers. In recent work, Pellizzoni and Caccamo [15] presented a scheme called M-CASH, which extends M-CBS by adding reclaiming techniques for reallocating processing capacity that becomes available when a server completes early. In this work, global EDF is assumed to be the top-level scheduler. Finally, in a somewhat different vein, Srinivasan *et al.* [19] proposed to solve the problem of integrating aperiodic jobs and HRT tasks by using Pfair-based algorithms as the top-level scheduler.

Unfortunately, none of the approaches above adequately solves the problem of interest to us. In particular, while the Pfair-based schemes can guarantee all real-time deadlines (soft or hard) without severely constraining overall utilization, this comes at the expense of higher runtime costs, as such algorithms tend to preempt and migrate jobs frequently. While such migrations are less costly on a multicore platform (due to the presence of on-chip shared caches), these algorithms also require that task execution costs be rounded to be an integral number of quanta, which can waste processing capacity. In all of the non-Pfair-based schemes discussed above, it is assumed that server jobs cannot miss their deadlines (even if the jobs of the tasks they encapsulate can). As noted earlier, caps on overall utilization must be enforced to ensure this. These caps can be quite restrictive—indeed, systems exist with total utilization of approximately $m/2$ that cannot be correctly scheduled (without missing deadlines) on m processors. For the kind of applications that have motivated our work, where only a small fraction of the workload is likely to have hard deadlines, insisting that all sever deadlines be met is overkill.

This realization marks the main departure of our work from prior efforts: we require that server deadlines not be missed *only* for those servers that encapsulate the HRT component of the workload. Other servers may miss their deadlines by bounded amounts. By allowing such misses, we are able to eliminate restrictive utilization caps, while still using a variant of global EDF. The fact that this is possible follows from recent work on global EDF. In particular, it has been shown that, when using global EDF to schedule sporadic real-time tasks on m processors, deadline tardiness is bounded, provided total utilization is at most m [9, 10, 20]. Furthermore, Devi and Anderson [10] have presented a variant of global EDF, called EDF-hl, that can guarantee zero tardiness to at most m tasks.

Proposed approach. We propose an extension of EDF-hl that does not restrict the number of HRT tasks. This extension, which we call EDF-HSB, creates up to m non-migratory servers to execute HRT tasks with zero tardiness. These servers are statically prioritized over the other servers in the system and can be provisioned independently of the periods of their clients. Each SRT task is serviced by a single server (or, equivalently, the task itself is scheduled directly by the top-level scheduler). Queued BE jobs are scheduled by additional SRT servers. The SRT servers may miss their deadlines, but by bounded amounts only. To make best use of dynamic slack when servers complete execution early, EDF-HSB uses a spare capacity redistribution method similar to CASH [6]. Jobs that finish early donate their unused capacity to a global capacity queue. Both SRT tasks that are likely to be tardy and BE jobs can receive such capacities to improve performance.

Summary of contributions. This paper breaks new ground in several ways. First, our server scheme is the first known to us that takes "SRT execution" as a first-class concept when scheduling servers. Second, our work is the first on implementing multiprocessor servers that is directed at what is likely to be the common case in practice: a system with components with different timing requirements for which the HRT component (if present) is relatively small. Third, our scheme is novel in that it is able to avoid caps on overall utilization (for the applications of interest to us), without resorting to more costly scheduling algorithms, such as Pfair-based schemes. Fourth, though others have previously considered reclamation schemes in the context of multiprocessors, we are the first to do so within a scheme where reclamation can be used to lower deadline tardiness in addition to improving BE responsiveness. Finally, to the best of our knowledge, no other multiprocessor server scheme has actually been implemented within a real operating system (though the designers of M-CASH have implemented both it and M-CBS within a real-time simulator [15]). In contrast, we have implemented EDF-HSB within Linux and thus have a real working framework.

Organization. The rest of this paper is organized as follows. In Secs. 2–4, respectively, we present needed definitions, describe EDF-HSB, and prove several important properties concerning it. Then, in Sec. 5, we discuss our implementation of EDF-HSB in Linux and an associated experimental evaluation. Finally, in Sec. 6, we conclude.

2 System Model

We consider the problem of scheduling a set τ of n fully preemptive, independent, sporadic real-time tasks concurrently with independent BE aperiodic jobs on a set of m identical multiprocessors with unit capacity. Each sporadic task $T_i = (e_i, p_i)$ is characterized by its worst-case *execution requirement*, e_i, its minimum inter-arrival time, or *period*, p_i, and its *utilization*, $u_i = e_i/p_i$. Each such task generates a sequence of jobs $T_{i,j}$, where $j \geq 1$. We denote the instant that a job becomes ready for execution as $r_{i,j}$, and require $r_{i,j} + p_i \leq r_{i,j+1}$. If the jobs of a sporadic task T_i are always released p_i time units apart, starting at time 0, then T_i is called a *periodic* task. If a job $T_{i,j}$ of a sporadic task executes for $e_{i,j} < e_i$ time units, then the resulting unused capacity, $e_i - e_{i,j}$, is referred to as *dynamic slack*. If such a job does not receive an allocation of $e_{i,j}$ time units before its implicit deadline $d_{i,j} = r_{i,j} + p_i$, then it is *tardy*. Note that, if a job of a sporadic task is tardy, then the release time of the next job of that task is not delayed.

All tasks and jobs are sequential, *i.e.*, the jobs of a sporadic task must execute in sequence, and each job can only execute on one processor at a time. We require all periods and deadlines to be some integral number of quanta (though execution costs can be non-integral). We assume this because actual hardware has inherent limits on timer resolution and reasonable overhead costs. For the purpose of the analysis, we consider preemption and migration costs to be negligible. However, we address these costs when discussing the issue of reclaiming spare capacities.

We assume in this paper that each sporadic task is either a *HRT task* or a *SRT task*. A HRT task may never miss a deadline. As noted earlier, the total utilization of all HRT tasks is assumed to be relatively small—more precisely, we require that it be possible to statically assign such tasks to processors such that no processor is overutilized. SRT tasks, on the other hand, may experience tardiness, but only by a bounded amount. This is sufficient to ensure that each SRT task receives a processor share matching its required utilization in the long term. As noted above, in addition to the jobs of real-time tasks, the system must process aperiodic *BE jobs*. These are unknown to the system before their arrival and no service guarantees can be ensured. BE jobs should be processed as fast as possible without compromising SRT and HRT guarantees.

3 Algorithm EDF-HSB

In this section, we present a new scheme, EDF-HSB, which meets the above requirements. EDF-HSB allows the system to be fully utilized and can flexibly deal with dynamic slack. The various mechanisms used in EDF-HSB are described in detail in later subsections. We begin with a general description of some of the underlying design choices.

EDF-HSB ensures the temporal correctness of HRT tasks by statically partitioning them among the available processors, and by encapsulating those assigned to each processor within a periodic *HRT server*, which executes only on that processor. These HRT servers offer three distinct advantages. First, because HRT tasks do not migrate, the analysis of their worst-case execution times is simplified. (It is worth noting that work on timing-analysis tools for multiprocessor platforms is in its infancy.) Second, our HRT servers require no over-provisioning, *i.e.*, such a server's utilization is simply the sum of the utilizations of its clients. Third, a HRT server's period can be sized independently of its client tasks, and thus its impact on SRT tasks and BE jobs can be adjusted freely. In contrast to our approach, in most (if not all) prior server approaches (*e.g.*, [5, 12, 13]), client tasks either may incur tardiness of up to the server's period, or the server must be over-provisioned in order to avoid client deadline misses (although it is important to acknowledge that these prior approaches were devised with different workloads in mind).

In EDF-HSB, the top-level scheduler schedules the HRT servers just described along with the other tasks in the system. These other tasks are prioritized against each other using a global EDF policy. Hence, they may migrate among the processors in the system. In addition, these other tasks may experience bounded deadline tardiness. Such tasks include both the SRT tasks that are part of the system to be scheduled and also a collection of sporadic *BE servers*, which are responsible for scheduling BE jobs. We will use the term "non-HRT task" when collectively referring to the set of SRT tasks and BE servers. When we use the term "job" in reference to a non-HRT task, and that task is a BE server, we are referring to an invocation of the server as scheduled by the top-level scheduler, and not a job that the server itself schedules.

Although EDF-HSB prioritizes HRT servers over other tasks, such servers may be considered ineligible for execution even when they have client tasks with unfinished jobs. The eligibility rules for these servers, given later, are defined so that non-HRT tasks can be given preference in scheduling when doing so would not cause HRT tasks to miss their deadlines.

Arriving aperiodic jobs are placed in a single global FIFO queue. When a BE server is scheduled for execution, it services jobs from this queue until either the queue empties or the server has exhausted its allocated budget (whichever occurs first). We assume that there are a total of m BE servers, so that arriving BE jobs can be scheduled in parallel to the extent possible. If the system experiences intervals of under-utilization, then the BE servers double as background servers.

At runtime, the performance of the system is further enhanced through a novel use of spare capacity redistribution to lessen tardiness for non-HRT tasks and to adapt quickly to load changes. For example, even if one processor is fully committed to serving HRT tasks, EDF-HSB can still use dynamic slack released on that processor to improve the performance of non-HRT tasks. The redistribution of slack is controlled by a heuristic. By using different heuristics, it is possible to tune EDF-HSB for various task loads. As discussed in Sec. 5, different heuristics can be defined depending on whether it is more important to lower BE job response times or to lower SRT task tardiness.

Example. Fig. 1 depicts an EDF-HSB schedule for a system of eight real-time tasks executing on three processors. Five of the tasks ($H_1, \ldots, H_5$) are HRT tasks, while the remaining three ($S_1, \ldots, S_3$) are SRT tasks. The HRT tasks are partitioned into two sets and assigned to HRT servers on Processors 1 and 2. The HRT component has a total utilization of $0.2+0.3 = 0.5$ and the SRT component has a total utilization of $1/3 + 1/3 + 1/3 = 1.0$. For the sake for illustration, the HRT server on Processor 2 is slightly over-provisioned ($1/3$ instead of 0.3) to generate a more interesting schedule. The remaining capacity of the system is distributed among three BE servers ($B_1, \ldots, B_3$). We assume that these servers are continually backlogged with BE jobs to schedule.

The HRT server on Processor 1 is not initially eligible (for reasons discussed later), but the one on Processor 2 is, so it is scheduled at time 0. This leaves two processors at time 0 on which to schedule two of $S_1, \ldots, S_3, B_1, \ldots, B_3$. S_2 and B_1 have the earliest deadlines, so they are selected for execution at time 0. We will consider other aspects of this schedule in detail later. For now, we turn our attention to describing the HRT server rules in detail.

3.1 Hard Real-Time Servers

Each HRT server S is a periodic task, with a maximum budget e_s, a current budget b_s, and a period p_s, that services a group of HRT tasks τ_s^H. The server's period can be chosen arbitrarily as long as its execution budget is scaled according to the total utilization of its client tasks, *i.e.*, $e_s = p_s \cdot \sum_{T_x \in \tau_s^H} u_x$. Short periods have less impact on non-HRT task tardiness but require more preemptions. Further, hardware constraints such as minimum quantum sizes may require rounding the budget to the next larger integral number of quanta, so a careless choice can cause a significant loss in allocatable utilization or an unnecessary increase in tardiness. In general, the optimal server period heavily depends on hardware properties such as preemption costs, time-keeping overhead, caching effects, *etc.* It is fairly straightforward to derive such a period analytically based on such parameters. However, due to space constraints, we omit this analysis from the paper.

As HRT servers are statically prioritized over other tasks by the top-level scheduler, the execution of a HRT server S with parameters e_s, p_s, and b_s is only governed by the replenishment and eligibility rules, S1–S6, below.

S1 The server executes whenever it is eligible, preempting any other task that may have been executing on its assigned processor before the server became active. It retains its budget when it suspends.

S2 The server is eligible if it has a positive budget, *i.e.*, $b_s > 0$, and at least one client job is eligible (see below).

S3 The server decrements its budget b_s by one for every time unit a client executes.

S4 Initially, the server's deadline d_s is set to p_s and its budget b_s is set to e_s. Every p_s time units, d_s is increased by p_s and b_s is reset to e_s.

The eligibility and priority of clients at time t is determined according to the following rules.

S5 A job $T_{k,j}^H$ generated by one of the tasks in τ^H is eligible if either **(a)** the job's deadline is earlier than the server's next deadline ($d_{k,j}^H < d_s$), or **(b)** the server has zero slack time in its current period (*i.e.*, $d_s - t = b_s$ at time t).

S6 The server selects the eligible client job with the earliest deadline for execution.

Note that rule S5b forces the server to always consume its complete budget as long as there are client jobs pending.

Example. Returning to the schedule in Fig. 1, at time 0, the deadline of the job $H_{5,1}$ (10) is earlier than that of its server H_2^S (15). Thus, according to Rule S5a, $H_{5,1}$ is eligible, and because H_2^S has sufficient budget, $H_{5,1}$ is scheduled on Processor 2 and completes. In contrast, at the same time, H_1^S has no client job with a deadline earlier than 10, it has non-zero slack, and there are competing non-HRT tasks. Thus, none of its client jobs are eligible and the server suspends itself, freeing Processor 1 for non-HRT tasks. H_1^S remains suspended until time 8, where it has zero slack and all of its jobs become eligible (Rule S5b). At that time, $H_{1,1}$ is the client job with the earliest deadline and thus is scheduled. It then completes after one time unit and $H_{2,1}$ is scheduled. Whenever a client job executes, its server's budget is decreased (Rule S3), and therefore H_1^S's budget is exhausted at time 10 and is replenished immediately. The update of H_1^S's deadline causes it to have non-zero slack again and it suspends until time 18. Note that, earlier in the schedule, at time 5, B_2 is able to execute as a background scheduler. This is because (by assumption) it is backlogged and a processor would otherwise be idle.

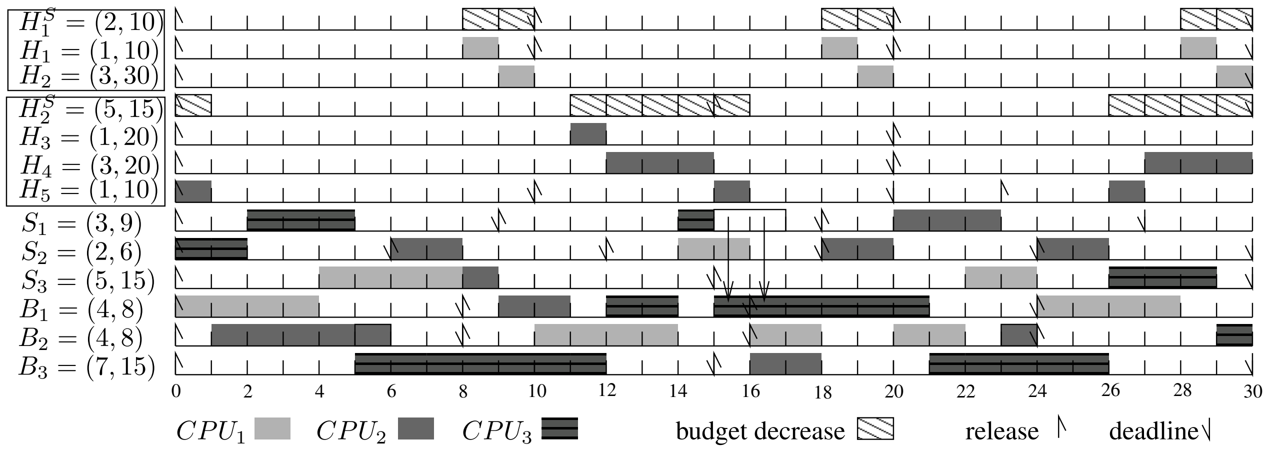

Figure 1: An example schedule illustrating EDF-HSB hard real-time server rules and capacity sharing.

On Processor 2, the jobs $H_{3,1}$ and $H_{4,1}$ are able to execute in the interval $[11, 15)$ by Rule S5b, and $H_{5,1}$ is able to execute at time 15 by Rule S5a. Later in the schedule, $H_{5,3}$ experiences a delayed sporadic release by three quanta until time 23. As $H_{5,3}$ is not eligible at the time of release, Processor 2 schedules B_2 as a background server.

In the above discussion, it has been assumed that an assignment of HRT tasks to processors exists. Such an assignment can be obtained using any of several bin-packing-based heuristics. However, as explained later in Sec. 4.2, some assignments may be more desirable than others from the standpoint of lessening non-HRT task tardiness.

3.2 Dynamic Slack Reclamation

Dynamic slack reclaiming works at the level of the top-level scheduler, so any references to the terms "job" and "task" in the ensuing discussion should be taken from the perspective of that scheduler, unless stated otherwise. While any job may produce dynamic slack, such slack may be consumed only by non-HRT tasks. An example of this can be seen at time 15 in Fig. 1, where the job $S_{1,2}$ completes two time units early, and these two units of capacity are consumed by the BE server B_1. Our goal is to redistribute slack so that tardiness is reduced for SRT tasks and BE response times are improved.

Our approach for meeting this goal is to use a modification of the CASH [6] and BACKSLASH [11] slack reclaiming algorithms. Similar to the multiprocessor extension M-CASH [15], EDF-HSB maintains a global queue of capacities sorted by deadline. These capacities are treated as "schedulable entities" by the top-level scheduler, and when a capacity is selected to execute, its processor time is (potentially) donated to some non-HRT task. Because of system overheads such as preemption and context-switching costs, not every amount of spare capacity can be consumed efficiently. Thus, we impose a lower bound of $q_{\min}$ on queued capacities. If a job creates a spare capacity of smaller size, it is not enqueued, but instead can be consumed only by the next non-HRT job that executes on the same processor. This allows small capacities to accumulate into usable chunks.

Our rules for managing spare capacities are as follows.

C1 A *capacity* $c = (q, d)$ is defined by a nonzero number of quanta q (perhaps non-integral) and a deadline d. Such a capacity c *expires* at time d.

C2 When a job $T_{i,j}$ completes at time $t < d_{i,j}$ after having received $e_{i,j} < e_i$ time units of service, it produces a capacity of $(\min(e_i - e_{i,j}, d_{i,j} - t), d_{i,j})$.

C3 A new capacity consisting of less than $q_{\min}$ quanta is assigned to the processor on which it was released and will be consumed by the next non-HRT job with a later deadline that executes on that processor, unless the processor idles beforehand. In the latter event, or if the capacity expires or is exhausted, it is disposed of.

C4 Non-expired capacities consisting of at least $q_{\min}$ quanta are stored in deadline order in a global queue.

C5 When a capacity is enqueued, it is immediately considered to be a "schedulable entity" by the top-level scheduler, *i.e.*, a capacity with a deadline at time d competes for processor time as if it were a SRT job with a deadline at time d. If the capacity is selected for execution, then it may execute only prior to time d and the processor time it receives can be consumed by any (single) non-HRT task with a current deadline at least d that may receive spare capacity. (In our implementation, we select such a task using a heuristic, as discussed later in Sec. 5.) A SRT task may receive spare capacity if its most recently released job has not completed execution. A BE server may do so if there are queued BE jobs. (A BE server may consume the capacity even if has executed for more than its worst-case execution cost in its current period.)

C6 When a non-HRT job J consumes a capacity, it is scheduled using the capacity's deadline and its own execution budget remains unaltered. If J completes or is preempted while consuming the capacity and if that capacity is neither expired nor exhausted, the leftover capacity is treated as a new spare capacity release according to C3 or C4.

C7 If a processor ever idles and the capacity queue is nonempty, then it dequeues the capacity with the earliest deadline and executes it without donating the resulting execution to any task. The processor continues to execute the capacity as long as it would otherwise be idle (and the capacity is neither exhausted nor expired).

One issue that we have glossed over above is that of determining when a server task can conclude that it has spare capacity to release. For a HRT server, this can be determined by having the server monitor the actual execution costs of its client tasks and releasing spare capacity that is created at that level. For a BE server, however, some application-specific knowledge of future aperiodic BE job arrivals would be required.

4 Correctness

We have three proof obligations, which are addressed in the following subsections: **(i)** no HRT task misses a deadline, **(ii)** tardiness is bounded for non-HRT tasks, and **(iii)** the reclamation of dynamic slack does not invalidate (i) and (ii).

4.1 Hard Real-Time Correctness

To establish HRT correctness, we first prove the following lemma concerning the use of (preemptive) *uniprocessor* EDF.

Lemma 1. *Let S denote a HRT server with maximum budget e_s, period p_s, utilization u_s, and client set τ_s^H, where $\sum_{T_x \in \tau_s^H} u_x \leq u_s \leq 1$. Further, let T_I be a periodic task with period $p_I = p_s$ and execution cost $e_I = p_s - e_s$. For any schedule of S under EDF-HSB with no dynamic slack reclaiming, there exists an EDF schedule of $\tau^* = \{T_I\} \cup \tau_s^H$ in which the allocations to tasks in τ_s^H are the same.*

Proof: We show that whenever S schedules one of its client tasks in EDF-HSB, the same scheduling choice can be made for τ^* under EDF. For a client job J of S to be scheduled under EDF-HSB, it must have been eligible via Rule S5. If J is eligible via Rule S5a, then its deadline is less than S's. Thus, in EDF, J's deadline is less than that of the current job of T_I (as S's deadlines and T_I's always coincide) and thus can be scheduled. If J's deadline is greater than S's current deadline, and J becomes eligible via Rule S5b, then S's slack is zero, which means that in the EDF schedule, the most recently released job of T_I has finished execution. Thus, J can be scheduled in the EDF schedule. □

Theorem 1. *In EDF-HSB with no dynamic slack reclaiming, no HRT job misses its deadline.*

Proof: Because the scheduling of a HRT server does not depend on the workload that competes with it on its assigned processor, this workload can be characterized via a fictitious task T_I as defined above. From Lemma 1 and the optimality of EDF on uniprocessors, the result follows. □

4.2 Tardiness

From the standpoint of the non-HRT tasks in the system, the HRT servers appear as a collection of up to m ordinary HRT periodic tasks that execute with higher priority and do not migrate. The EDF-hl algorithm of Devi and Anderson [10] mentioned in the introduction is similar: in that algorithm, a collection of SRT sporadic tasks is scheduled together with up to m HRT sporadic tasks that execute with higher priority. Because there are at most m such tasks, they can always be scheduled without migration. Devi and Anderson showed that in EDF-hl, SRT task tardiness is bounded. In this regard, our scheme differs from EDF-hl only in that our HRT servers may execute in a non-work-conserving manner due to the eligibility rules that we use. In particular, we allow such a server to retain a portion of its current budget without being scheduled until later in its current period. It can be shown, however, that this non-work-conserving behavior does not cause tardiness to become unbounded. In particular, letting τ_L denote the set of non-HRT tasks, τ_H denote the set of HRT servers, E_L denote the m largest SRT execution costs, U_L denote the m largest utilizations, $e_{\min}$ denote the smallest worst-case execution cost, $e_{\max}$ denote the maximum worst-case execution cost, and $u_{\max}^L$ denote the largest non-HRT utilization, we have the following theorem.

Theorem 2. *In EDF-HSB with no dynamic slack reclaiming, the tardiness of any task T_k in τ_L is at most $e_k + \min(x_1, x_2)$, where*

$$x_1 = \frac{E_L - e_{\min} + 2 \cdot \sum_{T_h \in \tau_H} e_h \cdot (1 - u_h)}{m - |\tau_H| - U_L},$$

$$x_2 = \frac{E_L + (|\tau_H| - 1) \cdot e_{\max} + 3 \cdot \sum_{T_h \in \tau_H} e_h \cdot (1 - u_h)}{m - \max(|\tau_H| - 1, 0) \cdot u_{\max}^L - U_L - \sum_{T_h \in \tau_H} u_h}.$$

While this theorem is a fairly minor extension of one proved by Devi and Anderson [10], it is not feasible to include its proof here, due to space constraints.

The tardiness bound above is similar in magnitude to others proved by Devi and Anderson [9, 10]. In practice, we have found that if task execution costs are worst-case values, then tardiness under global EDF is *substantially* lower than implied by these bounds. This suggests that the bounds are not tight.[3] Furthermore, with a worst-case provisioning, using dynamic slack to lower tardiness may not be necessary. The experiments given in Sec. 5 confirm this. However, if SRT tasks are provisioned using average-case execution costs, then tardiness can be much higher due to temporary overloads that may occur when a job's actual execution cost exceeds its average cost. All of our results can be applied without modification assuming such an average-case provisioning if such overloads are handled by requiring any overrunning job to finish execution by using time allocated to future jobs of the same task. Such an approach has the advantage that an overrunning job does not negatively impact the jobs of other tasks. However, with this approach, tardiness can be higher, since an overrunning job may actually execute as several jobs. In such a system, using dynamic slack to lower tardiness could be advantageous.

The above expressions reveal that the assignment of HRT tasks to processors (which determines the HRT server utilizations) may impact tardiness (as given by the bounds) for the tasks in τ_L. While an optimal assignment may be difficult to obtain, generally speaking, the bounds are lessened by an assignment that fully utilizes $k < m$ processors for HRT processing and balances the remaining HRT workload on the other processors (so that the HRT server utilization on these processors is approximately the same). Assuming (as we do here) that HRT tasks have fairly small utilizations, the first-fit heuristic will tend to nearly completely utilize some subset of the processors, while the worst-fit heuristic will tend to balance the load. An overall approach can be obtained by combining these heuristics and assessing tardiness for different values of k and the assignments produced.

4.3 Dynamic Slack Reclamation

We now show that dynamic slack reclamation does not compromise real-time correctness.

Theorem 3. *The dynamic slack reclamation mechanism of EDF-HSB does not invalidate the timing guarantees made in Theorems 1 and 2.*

Proof sketch. HRT correctness is straightforward: A HRT server only releases spare capacities and never consumes any. Given that HRT servers are prioritized over all non-HRT tasks, the manner in which non-HRT jobs consume spare capacities does not affect the scheduling of HRT jobs.

As for the tardiness of non-HRT jobs, small capacities that are dealt with via Rule C3 merely cause the consuming job to consume less of its own execution budget, which cannot increase tardiness. As for larger capacities that are dealt with via Rule C5, if such a capacity is donated by a non-HRT job, then the reasoning is also straightforward, due to Rules C5 and C6: the donated capacity is scheduled essentially as it would have been had it not been donated and instead were included as part of the execution of the donating job.

The only relatively tricky case is the situation where a HRT server donates a larger capacity. This is trickier because the donated capacity would have been executed at a higher priority level had it not been donated, but once donated, it competes for scheduling like a SRT job and is prioritized by its deadline. This change in the status of the donated capacity can cause it to shift later in time in the schedule. However, it will shift later only if a portion of the execution of some non-HRT job correspondingly shifts forward. Such modifications to the schedule cannot cause tardiness to increase. (Shifting some piece of computation c of a job of a task T to the future could potentially cause a problem if this causes a cascade of other future shifts. However, this could only happen if c is shifted so that it would execute concurrently with some portion of future jobs of T, which would cause future jobs to shift forward. Since a capacity can only be consumed prior to its deadline, this cannot happen in our case.) $\square$

5 Implementation

To assess the effectiveness of EDF-HSB in scheduling the kinds of workloads of interest to us, we implemented it within the Linux 2.6.9 kernel configured to run on a symmetric multiprocessor (SMP) architecture.[4] Our particular development platform is an SMP consisting of four 32-bit Intel(R) Xeon(TM) processors running at 2.70 GHz, with 8K instruction and data caches, and a unified 512K L2 cache per processor, and 2 GB of main memory. We implemented EDF-HSB by modifying a previously-developed system by our group called LITMUS$^{\mathrm{RT}}$ [8], which extends the base Linux kernel to allow different multiprocessor real-time scheduling algorithms to be utilized as plug-in components. Unfortunately, the previous LITMUS$^{\mathrm{RT}}$implementation did not support hierarchical scheduling, so implementing EDF-HSB within it actually required significant effort.

We do not have sufficient space to include an extensive discussion of how we modified LITMUS$^{\mathrm{RT}}$to support EDF-HSB, so we give here only a brief overview of some aspects of the implementation. HRT and BE servers were realized as accounting abstractions inside the kernel, with actual tasks implemented as Linux tasks. To create and modify servers, a new generic system call `sched_setup` was introduced that allows scheduler plugins to be configured similar to the standard system call `ioctl`. To allow for detailed analysis of the system's behavior, we implemented a new tracing facility. A sufficiently large ring-buffer was allocated for each CPU and exported as a character device to user space. During

[3]The bound given by Valente and Lipari [20] for global EDF is much smaller, but the proof of their bound was found to be in error.

[4]While the advent of multicore platforms is one of the motivating factors that led to our research, we currently do not have such a platform in our lab. We plan to port EDF-HSB to such a platform in the near future.

real-time mode, scheduling events such as new task arrivals, tasks blocking for I/O, and preemptions were recorded together with a timestamp in the ring-buffer of the CPU where they occurred. As each CPU has its own ring-buffer, there is no lock contention and the tracing overhead is reasonably small. The exported trace data was saved in user space for detailed offline analysis. This allowed us to perform in-depth trace analysis that would be too expensive to do online in kernel space.

5.1 Experiments

We tested the effectiveness of four schemes in scheduling a test workload consisting of HRT, SRT, and BE components. The four tested schemes were global EDF (each HRT and SRT task is treated as an ordinary EDF-scheduled task; each arriving BE job is scheduled as a background job), global EDF with BE servers (like global EDF, except that BE jobs are processed by BE servers as described in this paper), EDF-HSB without capacity sharing, and EDF-HSB with capacity sharing. We also measured BE job response times in an idle system.

We considered several test workloads in our work, one of which is discussed here. It consisted of eight synthetic HRT tasks with a total utilization of 0.8, fourteen synthetic SRT tasks with a total utilization of 2.76, and four HRT tasks on Processor 4 dedicated to flushing the trace ring buffers to disk. The remaining static capacity of 0.4 was assigned to four BE servers. The task-set composition is summarized in Table 1.

To evaluate the effectiveness of capacity sharing, we configured the HRT and SRT tasks to release spare capacity according to a Gaussian distribution. This method has been employed previously by Lin *et al.* [11]. Each real-time task was configured to execute for 75% of its worst-case execution time plus an additive term (which may be negative) that was obtained via a normal distribution with a mean of zero 0ms and variance of 20ms. The resulting actual execution time was limited to be within 50% to 100% of the corresponding task's worst-case execution time. The BE component was simulated using ten synthetic job generators that generate a sequence of jobs that require about 3ms to complete. Job releases were separated by an amount taken from a normal distribution with a mean of 100ms and variance of 40ms limited to lie with [0ms, 200ms].

Spare capacity can be used to improve BE response times or to lessen the tardiness of SRT tasks. As it is generally not possible to know in advance whether a task is going to be tardy (this would require knowledge of future job arrivals), a heuristic is needed to select tasks that are likely to be tardy. We implemented three different heuristics. The first heuristic compares a task's slack time with its remaining execution requirement. When the slack time is less than its execution requirement, it is considered to be in danger of being tardy.

HRT server on CPU 1	SRT component
3 x (3, 100)	2 x (12, 33)
2 x (2, 50)	2 x (7, 100)
1 x (64, 300)	2 x (50, 100)
1 x (130, 600)	2 x (25, 100)
1 x (200, 1000)	1 x (6, 33)
	1 x (6, 33)
HRT server on CPU 4	1 x (21, 150)
4 x (1, 100)	1 x (80, 400)
	1 x (168, 600)
BE servers	1 x (200, 1000)
4 x (10, 100)	

Table 1: The task set used to obtain the results given in Figs. 2, 3, and 4.

The second heuristic considers a task likely to be tardy if one of its last five jobs has been tardy. The third heuristic is a null heuristic that is used as a baseline. It never considers SRT tasks to be eligible for spare capacities before they are tardy. In each heuristic, SRT tasks are given preference to BE servers for consuming spare capacity. Of course, variants of these heuristics can be obtained in which BE servers are sometimes or always given preference.

In each of the four schemes we tested, *no real-time task (hard or soft) ever missed a deadline*. As a result, all spare capacity was donated to BE servers. The lack of any deadline misses is likely a consequence of the fact that, as noted earlier, global EDF rarely produces significant tardiness in a system provisioned assuming worst-case execution costs (as done for our test system). In future work, we plan to experimentally evaluate average-case-provisioned systems in which significant tardiness may occur. Even though no deadlines were missed in the two global EDF variants we tested, it is important to note that it is generally not possible to analytically *guarantee* that deadlines will not be missed under these schemes (without severely constraining overall utilization).

As no deadlines were missed in any of the tested schemes, the main metric that we used in comparing them was BE responsiveness. We ran each tested scheme for three minutes and then analyzed the resulting scheduler event traces for average and maximum BE response times as well as the response-time distribution. The results so obtained are described below.

5.2 Results

Our results are shown in several different ways in Figs. 2-4. Fig. 2 shows the BE job response-time distribution for each tested scheme. As can be seen here, response times under EDF are significantly worse than in the other schemes, and those under EDF-HSB with capacity sharing are significantly better. Interestingly, EDF with BE servers and EDF-HSB without capacity sharing have almost identical distributions. However, under these two schemes, HRT deadlines can be *guaranteed* only under EDF-HSB. Note also that capacity sharing significantly impacts performance under EDF-HSB.

In Fig. 3, the average and worst-case BE response time is

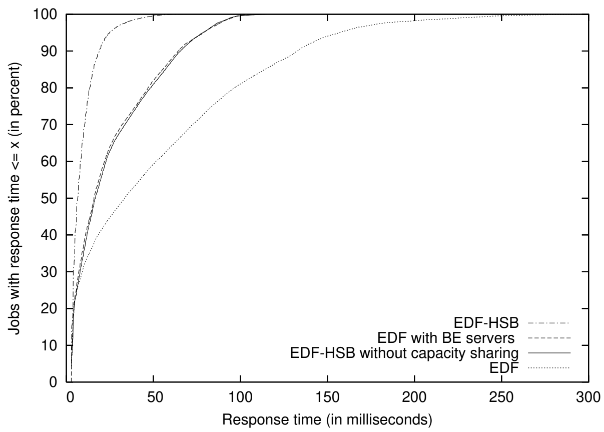

Figure 2: BE job response-time distribution for each tested scheme.

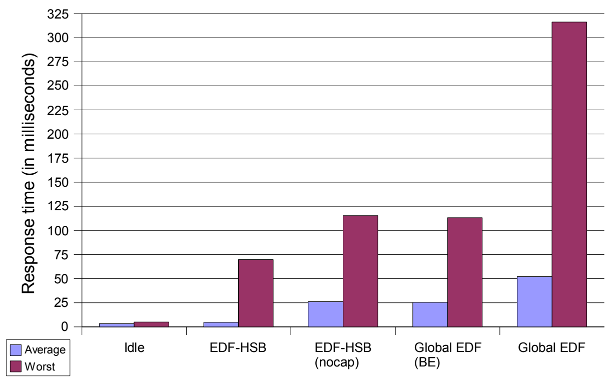

Figure 3: Average and worst-case BE job response times under each tested scheme.

shown for each scheme and also for an idle system. Interestingly, the *worst-case* response time under EDF-HSB with capacity sharing is reasonably close to the *average-case* response time under EDF. Note also that the average-case response time under EDF-HSB is quite close to that of an idle system.

Finally, in Fig. 4, the recorded response times under each scheme are given as scatter plots. Comparing inset (d) to insets (a)–(c), we see that rather long response times are possible without capacity sharing and (for this tested system) these long responses are completely eliminated by capacity sharing.

6 Conclusion

Our focus in this paper has been highly heterogeneous multiprocessor workloads that may consist of HRT, SRT, and BE components, where the HRT component is relatively small. We expect such workloads to become rather common as multicore platforms become more ubiquitous. Our goal has been to devise a scheme for supporting such workloads that does not require severe restrictions on overall utilization and that performs well in practice. The proposed scheme, EDF-HSB, meets this goal. We base this conclusion both on the formal analysis of EDF-HSB presented in Sec. 4 and the experimental validation discussed in Sec. 5. To our knowledge, EDF-HSB is the first real-time multiprocessor server scheme that does not require severe utilization restrictions and that has actually been implemented within a real operating system.

Numerous avenues for further research exist. These include adding support for synchronization to EDF-HSB, implementing some of the multiprocessor server schemes proposed by others and experimentally comparing them to EDF-HSB, and considering other variants of EDF-HSB in which some of the tradeoffs in system design discussed in this paper are resolved differently. One particularly interesting variant is one where nonpreemptive EDF is used to schedule HRT tasks. Our interest in this variant is motivated by the fact that timing analysis is much simpler if jobs are executed nonpreemptively. Another interesting open question is whether the tardiness bounds we have established can be tightened. Finally, as noted in [7], large multicore systems are likely to have hierarchical cache layouts, and in such systems, a scheduling approach that mixes aspects of partitioning and global scheduling might be preferable. In particular, while task migrations within a cluster of cores that share some lower level cache might be acceptable, migrations among processors that are "far apart" in the cache hierarchy may be too expensive. It would be interesting to extend EDF-HSB to take into account both cache asymmetry such as this and also processor asymmetry, which arises when different cores have different functional characteristics.

References

[1] S. Baruah, J. Goossens, and G. Lipari. Implementing constant-bandwidth servers upon multiprocessor platforms. In *Proceedings of the IEEE International Real-Time and Embedded Technology and Applications Symposium*, pages 154–163, 2002.

[2] S. Baruah and G. Lipari. A multiprocessor implementation of the total bandwidth server. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, 2004.

[3] A. Bechini and C. Prete. Performance-steered design of software architectures for embedded multicore systems. *Software-Practice and Experience*, 2002.

[4] S. Bisson. Azul announces 192 core Java appliance. http://www.itpro.co.uk/servers/news/99765/azul-announces-192-core-java-appliance.html, December 2006.

[5] L. A. G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-time Systems Symposium*, pages 3–13, 1998.

[6] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pages 295–304, 2000.

[7] J. Calandrino, J. Anderson, and D. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms.

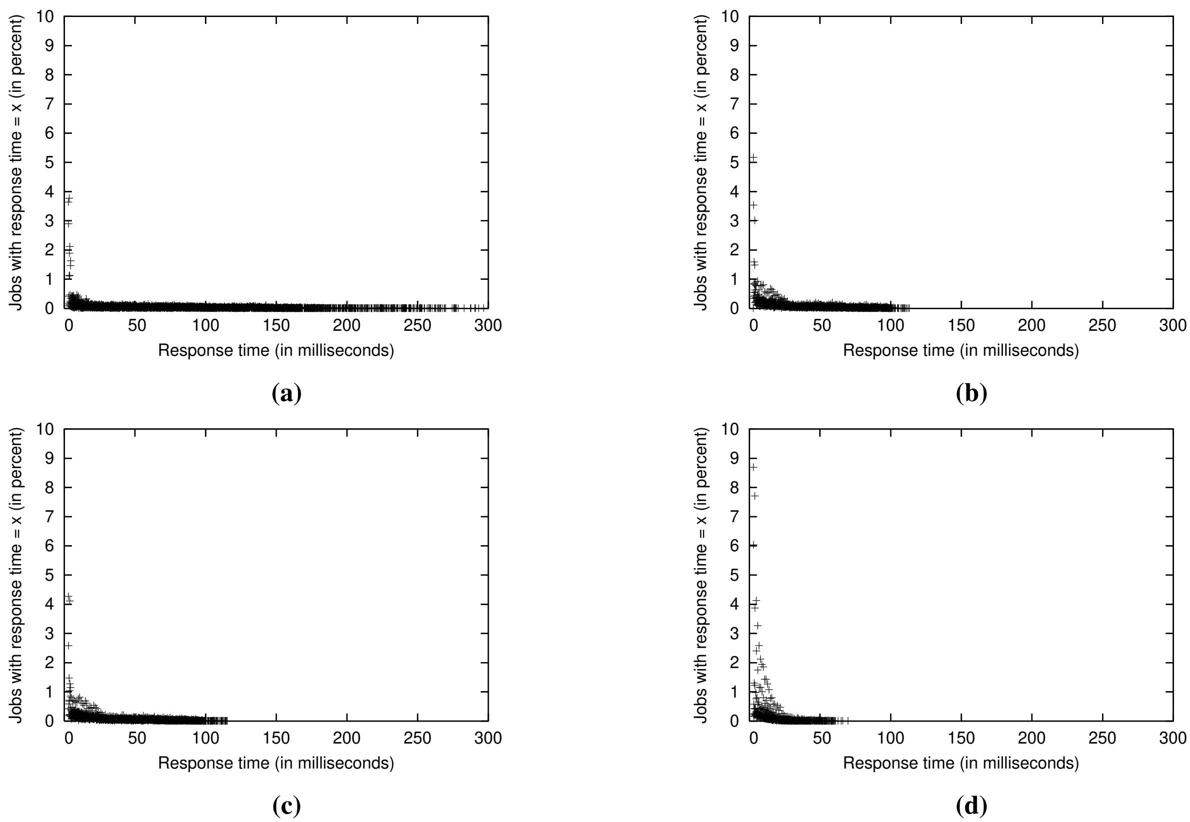

Figure 4: Scatter plots of response times under **(a)** EDF, **(b)** EDF with BE servers, **(c)** EDF-HSB without capacity sharing, and **(d)** EDF-HSB with capacity sharing.

In *Proceedings of the 19th EuroMicro Conference on Real-Time Systems*, (these proceedings), 2007.

[8] 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 IEEE International Real-Time Systems Symposium*, pages 111–123, 2006.

[9] U. Devi and J. Anderson. Tardiness bounds for global EDF scheduling on a multiprocessor. In *Proceedings of the 26th IEEE Real-time Systems Symposium*, pages 330–341, 2005.

[10] U. Devi and J. Anderson. Flexible tardiness bounds for sporadic real-time task systems on multiprocessors. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, 2006 (on CD ROM).

[11] C. Lin and S. Brandt. Improving soft real-time performance through better slack reclaiming. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 3–14, 2005.

[12] G. Lipari and S. Baruah. Greedy reclaimation of unused bandwidth in constant-bandwidth servers. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 193–200, 2000.

[13] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 151–158, 2003.

[14] Novell, Inc. SUSE Linux Enterprise Real Time. http://www.novell.com/products/realtime/, 2006.

[15] R. Pellizzoni and M. Caccamo. The M-CASH resource reclaiming algorithm for identical multiprocessor platforms. Technical Report UIUCDCS-R-2006-2703, University of Illinois at Urbana-Champaign, 2006.

[16] R. Rajkumar. Resource Kernels: Why Resource Reservation should be the Preferred Paradigm of Construction of Embedded Real-Time Systems. Keynote talk, 18th Euromicro Conference on Real-Time Systems, Dresden, Germany, 2006.

[17] M. Spuri and G. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the 15th IEEE Real-time Systems Symposium*, pages 228–237, 1994.

[18] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. *Journal of Computer and System Sciences*, 72(6):1094–1117, September 2006.

[19] A. Srinivasan, P. Holman, and J. Anderson. Integrating aperiodic and recurrent tasks on fair-scheduled multiprocessors. In *Proceedings of the 14th Euromicro Conference on Real-time Systems*, pages 19–28, 2002.

[20] P. Valente and G. Lipari. An upper bound to the lateness of soft real-time tasks scheduled by EDF on multiprocessors. In *Proceedings of the 26th IEEE Real-time Systems Symposium*, pages 311–320, 2005.