A Hierarchical Multiprocessor Bandwidth Reservation Scheme with Timing Guarantees*

Hennadiy Leontyev and James H. Anderson
Department of Computer Science, The University of North Carolina at Chapel Hill

Abstract

A multiprocessor scheduling scheme is presented for supporting hierarchical containers that encapsulate sporadic soft and hard real-time tasks. In this scheme, each container is allocated a specified bandwidth, which it uses to schedule its children (some of which may also be containers). This scheme is novel in that, with only soft real-time tasks, no utilization loss is incurred when provisioning containers, even in arbitrarily deep hierarchies. Presented experiments show that the proposed scheme performs well compared to conventional real-time scheduling techniques that do not provide container isolation.

1 Introduction

In the Linux community, two recent developments have occurred that are of relevance to real-time software design processes. The first is the introduction of "real-time" features such as high-resolution timers, priority inheritance, and shortened non-preemptable sections in mainline Linux (in versions 2.6.16 to 2.6.22). The second is the upcoming introduction (in version 2.6.24) of mechanisms for supporting *container hierarchies* [14, 10, 13]. Containers are an abstraction that allows different task groups to be isolated from one another (mainly, by providing different name spaces to different task groups for referring to tasks, files, *etc.*). Containers are seen as a lightweight way to achieve many of the benefits provided by virtualization, without the expense of hosting multiple operating systems. From the standpoint of scheduling, containers are similar to various "server" abstractions considered in the real-time-systems literature.

These Linux-related developments are happening at a time when multiprocessor platforms are becoming increasingly common. This is partly due to the advent of multicore technologies as an alternative to single-core chip designs. Additionally, reasonably-priced "server class" multiprocessors have been available for some time now. These hardware-related developments are profound, because they

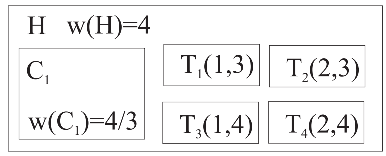


Figure 1. A host container H that encapsulates another container C_1 and four real-time tasks $T_1, \ldots, T_4$. Some of the notation used in this figure is explained in later sections.

mean that multiprocessors are now a "common-case" platform that software designers must deal with.

Motivated by these trends, we consider in this paper the problem of efficiently scheduling arbitrary real-time container hierarchies on a multiprocessor platform. Unlike most prior related efforts (see below), we are mainly interested in supporting *soft* timing constraints. This is partly due to our interest in Linux: while there is much interest in using Linux to support soft real-time workloads, Linux is not a real-time operating system and thus cannot be used to support "true" hard timing constraints. In addition, there is growing awareness in the real-time-systems community that, in many settings, soft constraints are far more common than hard constraints [19]. If hard constraints do exist, then ensuring them *efficiently* on most multiprocessor platforms is problematic anyway, due to a lack of effective timing-analysis tools for determining task execution costs. (While timing analysis is needed for soft real-time systems as well, less-accurate empirically-derived costs often suffice in such systems.)

The problem. For our purposes, a *container* is a scheduling abstraction. Containers are organized hierarchically in a tree. A container may have as children other containers or tasks, as seen in the example in Fig. 1. (In Linux, the container hierarchy may change dynamically; we defer consideration of dynamic changes to future work.) Each real-time task is assumed to be sporadic (see Sec. 2) and is either hard or soft: hard tasks cannot miss their deadlines, while soft tasks can. However, misses in the latter case may be by bounded amounts only. Associated with each container is a specified *bandwidth*, which denotes the fraction of the overall (multiprocessor) system's capacity to which it is entitled. When a container receives processor time, it allocates that

*Work supported by grants from Intel and IBM Corps., by NSF grants CNS 0408996, CCF 0541056, and CNS 0615197 and by ARO grant W911NF-06-1-0425.

time to one of its children. Our goal is to devise a scheme for performing such allocations throughout the containment hierarchy. Although we do allow for the presence of hard real-time tasks, we implicitly assume that they are few in number. That is, our main objective is to ensure that allocations are performed efficiently when most (or all) tasks are soft.

Of course, one way to meet this objective is by simply viewing all timing constraints as hard. However, in a container hierarchy, this will result in significant utilization loss. In particular, the schedulability of the tasks within a container depends on the processing capacity allotted to that container—the *supply*—and the processing capacity required by the tasks within the container—the *demand*. Very loosely speaking, verifying schedulability involves showing that demand (over some time interval of interest) cannot exceed supply. When timing constraints are hard, supply and demand are characterized using functions that cause supply to be under-estimated and demand to be over-estimated. The net effect is that, at each level of a containment hierarchy, some non-negligible amount of overall utilization is lost. The deeper the containment hierarchy, the greater the loss. In fact, the overall loss can be so great, unrestricted hierarchical containment simply becomes untenable.

Prior work. As noted earlier, the notion of a "container" as considered in this paper is more commonly called a "server" in the real-time-systems literature. Server-based abstractions were first considered in the context of uniprocessor systems, and a number of schemes intended for such systems have been proposed (too many for us to cite, due to a lack of space). Several multiprocessor schemes that are extensions of prior uniprocessor schemes have also been proposed [4, 5, 18]. However, in all of these schemes, it is assumed that all task deadlines are hard. Systems that may also have tasks with soft deadlines have been considered very recently [6]. In addition, several Pfair-based multiprocessor server schemes have been proposed, again, mostly for systems with only hard deadlines [1, 11, 17, 21]. In all of the work cited so far, only two-level containment hierarchies are considered. Moreover, the Pfair-based schemes just cited are subject to higher runtime overheads than other schemes, due to the fact that Pfair scheduling algorithms may preempt and migrate tasks often. The only prior work of which we are aware in which multi-level containment hierarchies are considered on multiprocessor platforms is a recent paper by Shin et al. [20]. However, as with most other prior work, only hard deadlines are considered in this paper. To the best of our knowledge, soft deadlines have not been considered before in scheduling-related research on supporting multi-level containment hierarchies on multiprocessors.

In the approach of Shin et al. [20], the global earliest-deadline-first algorithm (GEDF) is used as the per-container scheduler. Under GEDF, tasks are scheduled from a single run queue and their jobs are prioritized on an earliest-deadline-first (EDF) basis. One interesting property of GEDF is that, under it, bounded deadline tardiness can be ensured for sporadic tasks without severely constraining overall utilization [8]. In recent work, we showed that the same is true for a wider class of global scheduling algorithms [12]. In this paper, we exploit these results to obtain a hierarchical scheme in which deadline tardiness is bounded for soft real-time tasks.

Contributions. Our main contribution is a new multiprocessor scheduling approach for multi-level container hierarchies in which both hard and soft sporadic real-time tasks can be supported. With hard real-time tasks, some utilization loss is incurred (which seems inevitable). However, in a system with only soft real-time tasks, no utilization loss is incurred (assuming that system overheads are negligible—such overheads will cause some loss in any scheme in practice). This statement is true, provided the goal is to schedule soft real-time tasks so that their tardiness is bounded, no matter how great the bound may be. Tardiness bounds can be reduced by constraining overall utilization, and such tradeoffs are discussed herein. In addition to presenting our overall scheme, we also present the results of experiments conducted to assess its usefulness. In these experiments, our scheme exhibited performance—in terms of both necessary processing capacity and tardiness—comparable to that of schemes that exhibit good performance but are oblivious to containers (and hence, do not provide any container isolation).

The rest of this paper is organized as follows. In Sec. 2, we present our system model. In Sec. 3, we formally characterize the "supply" available to a container and propose a container scheduling scheme. In Secs. 4 and 5, we present methods for checking the schedulability of real-time tasks within a container and for computing the supply available to its child containers (if any). In Sec. 6, we present our experimental results. We conclude in Sec. 7.

2. System Model

In order to support the scheduling of containers within an arbitrary hierarchy, it suffices to consider the problem of scheduling a single container H on a set of $M(H)$ unit-speed processors, where some processors may not be available for execution during certain time intervals. The set of child containers and real-time tasks encapsulated in H is referred to as $succ(H)$. (Non-real-time tasks could be contained as well, but we do not consider such tasks in this paper.) At any time, the container may be scheduled on several available processors. When the container is scheduled, some of its children are selected for execution using some internal scheduling policy.

2.1. Sporadic Task Model

The set of real-time tasks encapsulated in the container H is denoted $\tau = \{T_1, \ldots, T_n\}$. In this paper, we assume such tasks are sporadic. Each sporadic task is invoked or *released* repeatedly, with each such invocation called a *job*. Associated with each such task T_i are two parameters, e_i and p_i: e_i gives the maximum *execution time* of one job of T_i, while, p_i, called the *period* of T_i, gives the minimum time between consecutive job releases of T_i. For brevity, we often use the notation $T_i = (e_i, p_i)$ to specify task parameters. The *utilization of task* T_i is defined as $u_i = e_i/p_i$, and the *utilization of the task system* τ as $U_{sum}(\tau) = \sum_{T_i \in \tau} u_i$.

The j^{th} job of task T_i, where $j \geq 1$, is denoted $T_{i,j}$. A task's first job may be released at any time $t \geq 0$. The release time of job $T_{i,j}$ is denoted $r_{i,j}$ and its (absolute) deadline $d_{i,j}$ is defined as $r_{i,j} + p_i$ (implicit deadlines). 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. When a job of a task misses its deadline, the release time of the next job of that task is not altered. However, at most one job of a task may execute at any time, even if deadlines are missed. Given these assumptions, if a task has bounded deadline tardiness, then its long-term allocation is proportional to its utilization. For *hard* real-time (HRT) tasks, we require that all deadlines are met, while for *soft* real-time (SRT) tasks, we require that deadline tardiness to be bounded (regardless of how high the bound may be).

In that which follows, we find it convenient to view a real-time task as a specialized container with no nested children that can be scheduled on at most one processor at any time and that has hard or soft deadlines.

2.2. Container Bandwidth

Each container H is characterized by its *bandwidth* $w(H) \geq 0$, which specifies the processing capacity to which it is entitled. For a real-time task T_i, we define $w(T_i) \triangleq u_i$. Since the containers in $succ(H)$ are scheduled when the parent container is scheduled, their allocation time cannot exceed that of H. Therefore, we require

$$w(H) \geq \sum_{C_j \in succ(H)} w(C_j). \qquad (1)$$

Example 1. In Fig. 1, a host container H with bandwidth $w(H) = 4$ encapsulates a child container C_1 with bandwidth $w(C_1) = 4/3$, two HRT tasks $T_1(1, 3)$ and $T_2(2, 3)$, and two SRT tasks $T_3(1, 4)$ and $T_4(2, 4)$.

Overview of our approach. In the following sections, we solve the problem described at the beginning of Sec. 2 via a decomposition into two subproblems, each of which can be solved by extrapolating from previously-published results. First, we split the bandwidth of each container, parent and child, into integral and fractional parts and argue that the integral parts can easily be dealt with. The fractional part of each child container is then handled by creating a special SRT *server task* with utilization equal to that fractional portion. This leads to our first subproblem, which is that of scheduling within the parent container, using the "supply" available to it, all child HRT and SRT tasks (where some of the SRT tasks may be server tasks). We then deal with any HRT tasks by encapsulating them within a new child container that schedules these tasks on an integral number of processors via a prior HRT scheduling scheme. This leaves us with our second subproblem, which is to schedule within the parent container a collection of SRT tasks. We solve this problem by exploiting prior results on using global scheduling algorithms to ensure bounded tardiness. So that our overall scheme can be applied inductively in a container hierarchy, we finish our analysis by characterizing the supply available to each child container.

3. Container Scheduling

The host container H receives processor time from $M(H)$ individual processors. We now further constrain the manner in which any container C receives processor time by assuming the following.

(P) At any time, a container C can be scheduled on $m(C) \triangleq \lfloor w(C) \rfloor$ or $M(C) \triangleq \lceil w(C) \rceil$ processors.

This restriction minimizes the execution parallelism available to C so that, for any interval of length Δ, C's allocation is within $[\lfloor w(C) \rfloor \Delta, \lceil w(C) \rceil \Delta]$. For real-time tasks, this restriction holds implicitly, because a real-time task T_i is scheduled on at most one processor at any time and $w(T_i) = u_i \leq 1$, so $\lceil w(T_i) \rceil = 1$ and $\lfloor w(T_i) \rfloor = 0$. We say that a processor is *fully available* to C, if it is dedicated exclusively to C. Given Restriction (P), we can assume that $m(C)$ processors are fully available to C.

As explained in detail later, there are two reasons for introducing Restriction (P). First, increasing the amount of supply parallelism (the number of available processors) restricts the maximum per-task utilization and the total system utilization if the long-term supply remains fixed. Second, maximizing the number of processors fully dedicated to C lessens deadline tardiness for any child real-time task. Intuitively, this is because such tasks are *sequential* and thus may leave processors unused if parallelism is increased too much.

Example 2. Consider a container H with bandwidth $w(H) = 4/3$ that encapsulates a task $T_1(5, 6)$, as shown in Fig. 2(a). Suppose that processor time is supplied as shown in Fig. 2(b) so that H occupies two processors for two time

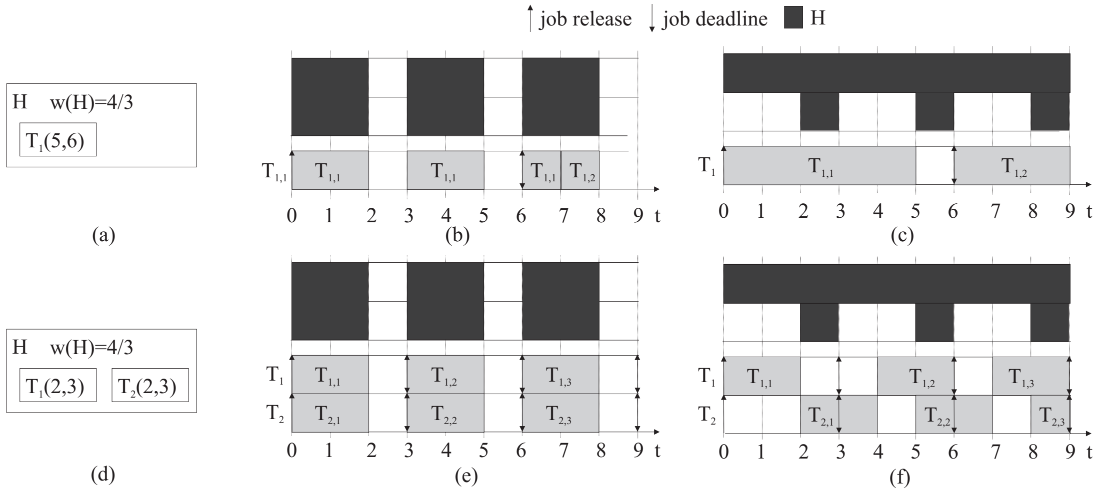


Figure 2. Comparison of supply parallelism in Examples 2 and 3.

units every three time units. The supply available to H is approximately $\frac{4 \cdot \Delta}{3}$ for any sufficiently long interval Δ. However, H does not execute during the interval $[2, 3)$, so Restriction (P) is violated, because $\lfloor w(H) \rfloor = \lfloor 4/3 \rfloor = 1$. Task T_1's jobs demand five execution units every six time units, but because they must execute sequentially, they can execute for only four time units every six time units. Thus, task T_1's tardiness can be unbounded. In the schedule in Fig. 2(c), container H also receives four execution units every three time units, but in contrast to Fig. 2(b), (P) is satisfied. Because one processor is fully available to H, task T_1 meets all of its deadlines.

As one may suspect, enforcing Restriction (P) may sometimes have negative consequences. Indeed, a task set with a large number of tasks may benefit from a larger number of available processors if all deadlines have to be met.

Example 3. Consider the container H from the previous example, except that it now encapsulates two real-time tasks $T_1(2, 3)$ and $T_2(2, 3)$, as shown in Fig. 2(d). In the schedule shown in Fig. 2(e), which is equivalent from container's perspective to that in Fig. 2(b), jobs of T_1 and T_2 meet their deadlines. However, in the schedule in Fig. 2(f), where Restriction (P) is enforced as in Fig. 2(c), job $T_{2,1}$ misses its deadline at time three because it cannot execute on two processors simultaneously during the time interval $[2, 3)$. Still, in this schedule, T_2's tardiness is only one time unit.

The two examples above illustrate that, while minimizing supply parallelism may negatively impact timeliness, it allows the widest range of loads to be scheduled with bounded deadline tardiness, which is in accordance with our focus on SRT tasks. In [20], mentioned earlier, the objective is instead to schedule HRT tasks. There, the alternative approach of maximizing supply parallelism is used.

We now develop a scheduling policy that enforces Restriction (P) for child containers assuming that it holds for the host container H. Given the latter, H is supplied time from $M(H)$ processors, where $m(H)$ processors are always available for scheduling $succ(H)$ and at most one processor is partially available.

A child container $C_i \in succ(H)$ must occupy at least $m(C_i)$ processors at any time. By (1), $w(H) \geq \sum_{C_i \in succ(H)} w(C_i)$, and hence, $m(H) = \lfloor w(H) \rfloor \geq \lfloor \sum_{C_i \in succ(H)} w(C_i) \rfloor \geq \sum_{C_i \in succ(H)} \lfloor w(C_i) \rfloor = \sum_{C_i \in succ(H)} m(C_i)$. Therefore, we can make $m(C_i)$ processors fully available to each child container $C_i \in succ(H)$ by using the $m(H)$ processors fully available to H. Note that, for containers with $w(C_i) < 1$ (including real-time tasks), $m(C_i) = \lfloor w(C_i) \rfloor = 0$. In any event, given this design decision, each child container C_i receives at least $m(C_i)\Delta$ units of time over an interval of length Δ.

If a child container C_i is not a real-time task and $m(C_i) < w(C_i)$, then it occasionally needs supply from an additional processor. For this, we construct a SRT periodic *server task* $S_i(e_i, p_i)$, where $u_i = e_i/p_i = w(C_i) - m(C_i) < 1$. (A *periodic* task is a special case of a sporadic task for which the parameter p_i represents an *exact* spacing between job releases.)

We denote the set of server tasks as $\tau^S = \{S_1, \ldots, S_n\}$. Jobs of these tasks are scheduled together with the jobs of encapsulated real-time tasks using the remaining $m(H) - \sum_{C_j \in succ(H)} m(C_j)$ fully available processors and at most one partially available processor. When task S_i's jobs are scheduled, an additional processor is available to container C_i. Because server task S_i is constructed only if $w(C_i) > m(C_i) = \lfloor w(C_i) \rfloor$, $\lceil w(C_i) \rceil = m(C_i) + 1 = M(C_i)$. Thus, container C_i always occupies $m(C_i)$ processors, and $M(C_i)$ processors are occupied when task S_i's job is scheduled. Thus, Restriction (P) is ensured for each child container.

Example 4. Consider container H from Example 1. For container C_1, one processor is reserved because $\lfloor w(C_1) \rfloor =$

$\lfloor 4/3 \rfloor = 1$. For this container, we also construct a SRT server task $S_1(1,3)$, so that $\lfloor w(C_1) \rfloor + e_1/p_1 = 1 + 1/3 = w(C_1)$. When jobs of S_1 are scheduled, an additional processor is available to container C_1, as shown in Fig. 3(b).

Let $\mathsf{HRT}(H)$ (respectively, $\mathsf{SRT}(H)$) be the set of HRT (respectively, SRT) tasks encapsulated in H. The remaining problem at hand, referred to as Subproblem 1, is that of scheduling tasks from the sets $\mathsf{HRT}(H)$, $\mathsf{SRT}(H)$, and τ^S on some number of fully available processors and at most one partially available processor.

4. Subproblem 1

To schedule the tasks in $\mathsf{HRT}(H)$, we encapsulate them into a child container C_{hrt} with integral bandwidth $w(C_{hrt}) = m(C_{hrt}) = M(C_{hrt})$. Applying (P) to C_{hrt}, $m(C_{hrt})$ processors must be reserved for this container.

The tasks in $\mathsf{HRT}(H)$ can be scheduled within C_{hrt} using a variety of approaches. Given our emphasis on SRT tasks, we simply use the partitioned EDF (PEDF) algorithm for this purpose, deferring consideration of other approaches to future work. Under PEDF, tasks are statically assigned to processors and each processor schedules its assigned tasks independently on an EDF basis. Assume that processor k is among the $m(C_{hrt})$ processors reserved for container C_{hrt} and let τ_k denote the set of sporadic tasks assigned to that processor. All task deadlines will be met on processor k if

$$U_{sum}(\tau_k) = \sum_{T_i \in \tau_k} u_i \leq 1, \tag{2}$$

which is a well-known uniprocessor EDF schedulability test [16]. This test, when applied in a multiprocessor system, presumes a given assignment of tasks to processors. Such an assignment (and correspondingly, the number of processors required for C_{hrt}) can be determined using any of various bin-packing heuristics. Further results concerning PEDF schedulability tests can be found in [3, 7, 16].

As mentioned earlier, HRT policies may introduce utilization loss. For PEDF, there exist task sets, for which the reserved processors could be underutilized. However, if HRT tasks are relatively few in number, such loss will likely be small, compared to the total utilization of SRT tasks.

Loss is incurred when creating C_{hrt} if its bandwidth (given by the number of processors required for it) exceeds the sum of the utilizations of the HRT tasks it contains. If this is the case, then (1) must be validated with the tasks in $\mathsf{HRT}(H)$ replaced by the container C_{hrt}. Note that, if we have a system with small number of processors, then it may not be possible to dedicate a processor for an HRT container as described above. In this case, it might be necessary for HRT and SRT tasks to execute on the same processor, which would require different analysis from that in this paper. Given our focus on SRT tasks, such analysis is beyond the scope of this paper. However, if a system is purely SRT, an arbitrarity deep hierarchy of SRT containters can be maintained even in the uniprocessor case.

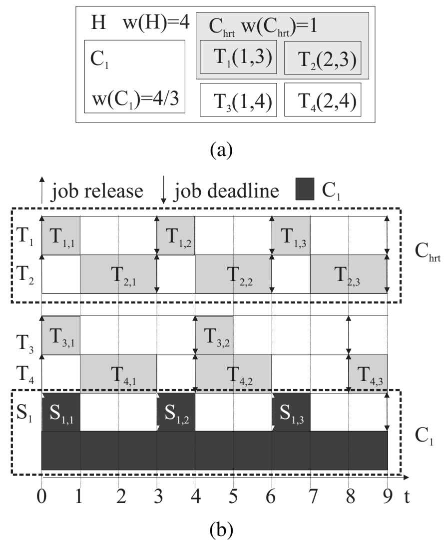


Figure 3. (a) Isolating HRT tasks. (b) A schedule with HRT in a separate container in Example 5.

Example 5. Consider again container H from Example 1. In our approach, we encapsulate the two HRT tasks $T_1(1,3)$ and $T_2(2,3)$ into a container C_{hrt}, as shown in Fig. 3(a). The total utilization of these two tasks is $U_{sum} = u_1 + u_2 = 1/3 + 2/3 = 1$. By (2), these two tasks will meet their deadlines if scheduled using uniprocessor EDF. We set $w(C_{hrt}) = 1$, so the container C_{hrt} will require one processor. The total bandwidth of container H's children is $\sum_{C_i \in succ(H)} w(C_i) = w(C_1) + w(C_{hrt}) + w(T_3) + w(T_4) = 4/3 + 1 + 1/4 + 2/4 = 37/12 < 4 = w(H)$, so (1) is satisfied. When scheduling the modified container H on $\lceil w(H) \rceil = 4$ processors, as shown in Fig. 3(b), one processor is reserved for the HRT container C_{hrt}, so that tasks T_1 and T_2 are scheduled on that processor. In Example 4, we reserved one processor for container C_1 and constructed the server task $S_1(1,3)$. Jobs of this server task are scheduled with the jobs of tasks T_3 and T_4 on the two remaining fully available processors.

Having dispensed with any HRT tasks, we can complete our solution to Subproblem 1 by devising a scheduling policy that ensures bounded tardiness for the remaining SRT tasks, some of which may be server tasks. These tasks are scheduled on M_s processors, of which $m(H) - \sum_{C_j \in succ(H)} m(C_j)$ are fully available and at most one is partially available. We refer to this last remaining subproblem as Subproblem 2.

5. Subproblem 2

In solving Subproblem 2, restrictions on supplied processor time are of relevance. Such restrictions can be dealt with using per-processor *supply or availability functions* [7].

Definition 1. (supply functions) The supply (or availability) function $\beta_k^l(\Delta)$, $R \rightarrow R$, provides a lower bound on the amount of processor time processor k can guarantee during any time interval of length Δ. This function is defined as

$$\beta_k^l(\Delta) = \max(0, \widehat{u_k} \cdot (\Delta - \sigma_k)), \tag{3}$$

where $\widehat{u_k} \in (0, 1]$ and $\sigma_k \geq 0$. $\widehat{u_k}$ is called the *processor bandwidth* and σ_k the *maximum blackout time*, because σ_k is the maximum interval when processor k may not provide any supply [9].

The following property is a straightforward application of the above definition.

(F) If processor k is fully available, then $\beta_k^l(\Delta) = \Delta$, $\widehat{u_k} = 1$, and $\sigma_k = 0$.

From the earlier statement of Subproblem 2, of the M_s processors under consideration, at most one is partially available. Thus, the supply from these processors can be described using M_s supply functions: $\beta_1^l(\Delta) = \max(0, \widehat{u_1}(\Delta - \sigma_1))$, where $0 < \widehat{u_1} \leq 1$, $\sigma_1 \geq 0$, and $\beta_k^l(\Delta) = \Delta$, for $2 \leq k \leq M_s$. We say that such a collection of functions is in *Minimum Parallelism* (MP) form.

Before continuing, note that if $M_s = 1$, i.e., all remaining SRT tasks are to be scheduled on one processor, then EDF can be used on that processor. If this processor is fully available, then tardiness will be zero for these tasks (due to the optimality of EDF), and if it is partially available, then it can be easily shown to be bounded, using real-time calculus [7], provided $U_{sum}(\mathsf{SRT}(H) \cup \tau^S) \leq \widehat{u_1}$. In the remainder of this section, we concentrate on the more interesting case, $M_s \geq 2$. In this case, our approach leverages some recent theoretical results, which we describe next.

5.1. Window-Constrained Scheduling

The problem of scheduling a set of sporadic SRT tasks on multiple processors with restricted supply was considered by us in [12]. In this work, a class of global scheduling policies that ensure bounded deadline tardiness was considered. This class of algorithms is described next.

Let τ be a set of sporadic SRT tasks scheduled on $M \geq 2$ processors, with supply functions $\beta_k^l(\Delta) = \max(0, \widehat{u_k}(\Delta - \sigma_k))$, where $1 \leq k \leq M$. Assume

$$U_{sum}(\tau) \leq \sum_{k=1}^{M} \widehat{u_k}, \tag{4}$$

i.e., the total system utilization is at most the total supplied bandwidth. Released jobs are placed into a single global ready queue. When choosing a new job to schedule, the scheduler selects (and dequeues) the ready job of highest priority. Priorities are determined as follows assuming that any ties are broken arbitrarily but consistently.

Definition 2. (prioritization functions) Associated with each released job $T_{i,j}$ is a function of time $\chi_{i,j}(t)$, called its *prioritization function*. If $\chi_{i,j}(t) < \chi_{k,h}(t)$, then the priority of $T_{i,j}$ is higher than the priority of $T_{k,h}$ at time t.

Definition 3. (window-constrained priorities) A scheduling algorithm's prioritization functions are *window-constrained* iff, for each task T_i, there exist constants $\phi_i \geq 0$ and $\psi_i \geq 0$ such that, for each job $T_{i,j}$ of T_i,

$$r_{i,j} - \phi_i \leq \chi_{i,j}(t) \leq d_{i,j} + \psi_i \tag{5}$$

holds at each time t where $T_{i,j}$ is pending.

GEDF is an example of a global algorithm with window-constrained priorities. In [12], a tardiness bound is established that applies to any window-constrained global scheduling algorithm. To state this bound, let

$$\rho = \max_{T_i \in \tau}(\phi_i) + \max_{T_i \in \tau}(\psi_i). \tag{6}$$

Further, let $U(\tau, y)$ $(E(\tau, y))$ be the set of at most y tasks from τ of highest utilization (execution cost), and let

$$E_L = \sum_{T_i \in E(\tau, M-1)} e_i \text{ and } U_L = \sum_{T_i \in U(\tau, M-1)} u_i. \tag{7}$$

Theorem 1. *(Proved in [12]) The tardiness of any task $T_k \in \tau$ under a window-constrained scheduling policy on $M \geq 2$ processors is at most $\max(x, \rho) + e_k$, where*

$$x = \frac{E_L + \max(A(\ell))}{\sum_{k=1}^{M} \widehat{u_k} - \max(H-1, 0) \cdot \max(u_\ell) - U_L}, \tag{8}$$

$$\begin{aligned} A(\ell) \quad = \quad & e_\ell \cdot \left(\sum_{k=1}^{M}(1 - \widehat{u_k}) - 1\right) + \sum_{k=1}^{M} \widehat{u_k} \cdot (\sigma + \sigma_k) \\ & + \sum_{T_k \in \tau \setminus T_\ell} \left(\left\lceil \frac{\psi_\ell + \phi_k}{p_k} \right\rceil + 1 \right) \cdot e_k \\ & + (M - 1 - max(H-1, 0) \cdot u_\ell) \cdot \rho, \end{aligned} \tag{9}$$

$\sigma = \max_{k \in [1..M]}(\sigma_k)$, and H is the number of processors with $\beta_k^l(\Delta) \neq \Delta$, provided the denominator of x is positive.

5.2. Minimizing the Tardiness Bound

Given the theorem stated above, we now argue in favor of Restriction (P) and show how enforcing this restriction affects the tardiness bound in the theorem. Consider the denominator of (8):

$$\sum_{k=1}^{M} \widehat{u_k} - \max(H-1,0) \cdot \max(u_\ell) - U_L. \tag{10}$$

The requirement for (10) to be positive implicitly restricts the maximum per-task utilization if $H > 1$, i.e., if two or more processors are partially available. Note also that the value of x is minimized if (10) is maximized. Suppose that the total supplied bandwidth $W = \sum_{k=1}^{M} \widehat{u_k}$ is fixed. Then, (10) will be maximized if either $\max(H-1,0) \cdot \max(u_\ell)$ or U_L or both are minimized. The value of U_L depends exclusively on task utilizations and the total number of processors M, as (7) suggests. Therefore, U_L will be minimized if the total number of processors M is minimized. The expression $\max(H-1,0) \cdot \max(u_\ell)$ is minimized if $H \le 1$, that is, at most one processor is partially available. Thus, if the total processor bandwidth W is fixed, then (10) is maximized by setting $M = \lceil W \rceil$ and having $\lfloor W \rfloor$ processors fully available. The bandwidth of at most one partially available processor (if any) is $\widehat{u_1} = W - \lfloor W \rfloor$. Referring back to Subproblem 2, we have the following.

Corollary 1. *Let $\tau = \mathsf{SRT}(H) \cup \tau^S$ be a set of sporadic SRT tasks scheduled on $M_s \ge 2$ processors with supply in MP form. Then, the tardiness of any task $T_k \in \tau$ under a window-constrained scheduling policy is at most $\max(x, \rho) + e_k$, where*

$$x = \frac{E_L + \max(A(\ell))}{M_s - 1 + \widehat{u_1} - U_L}, \tag{11}$$

$$\begin{aligned} A(\ell) &= e_\ell \cdot (-\widehat{u_1}) + (2\widehat{u_1} + M_s - 1)\sigma_1 \\ &+ \sum_{T_k \in \tau \setminus T_\ell} \left(\left\lceil \frac{\psi_\ell + \phi_k}{p_k} \right\rceil + 1 \right) \cdot e_k + (M_s - 1) \cdot \rho. \end{aligned}$$

Proof. In the case of supply in MP form, at most one supply function $\beta_1^l(\Delta)$ may differ from Δ. Thus, $H \le 1$. By (F),

$$\forall 2 \le k \le M_s : \sigma_k = 0, \widehat{u_k} = 1. \tag{12}$$

Thus,

$$\sigma = \max_{k \in [1..M_s]} (\sigma_k) = \sigma_1, \tag{13}$$

$$(\sum_{k=1}^{M_s} (1 - \widehat{u_k}) - 1) = (1 - \widehat{u_1} - 1) = -\widehat{u_1}, \tag{14}$$

$$\begin{aligned} \sum_{k=1}^{M_s} \widehat{u_k} \cdot (\sigma + \sigma_k) &= \widehat{u_1}(\sigma + \sigma_1) + \sum_{k=2}^{M_s} \widehat{u_k} \cdot (\sigma + \sigma_k) \\ &= \{\text{by (12) and (13)}\} \\ &\quad 2\widehat{u_1}\sigma_1 + (M_s - 1)\sigma_1 \\ &= (2\widehat{u_1} + M_s - 1)\sigma_1. \end{aligned} \tag{15}$$

Setting $H \le 1$ and (12)–(15) into (8) and (9), we get

$$x = \frac{E_L + \max(A(\ell))}{M_s - 1 + \widehat{u_1} - U_L},$$

$$\begin{aligned} A(\ell) &= e_\ell \cdot (-\widehat{u_1}) + (2\widehat{u_1} + M_s - 1)\sigma_1 \\ &+ \sum_{T_k \in \tau \setminus T_\ell} \left(\left\lceil \frac{\psi_\ell + \phi_k}{p_k} \right\rceil + 1 \right) \cdot e_k + (M_s - 1) \cdot \rho. \end{aligned}$$

The denominator of x in (11) is always positive, because $U_L < U_{sum} \le \sum_{k=1}^{M_s} \widehat{u_k} = M_s - 1 + \widehat{u_1}$. The corollary immediately follows. □

If GEDF is used for SRT tasks, then the tardiness bound can be further tightened by setting $A(\ell)$ in (11) to $e_\ell \cdot (-\widehat{u_1}) + (2\widehat{u_1} + (M_s - 1))\sigma_1$. Further, if all $M_s \ge 2$ processors are fully available, a HRT GEDF schedulability test [2] can be applied to τ before calculating tardiness bounds. If this test passes, then maximum tardiness is zero.

Note that Corollary 1 only requires that (4) holds (with M replaced by M_s). That is, bounded tardiness can be ensured with no utilization loss.

5.3. Computing Next-Level Supply

The remaining issue is to compute the supply of each child container in MP form, so that our analysis can be applied inductively in a container hierarchy. If a server task $S_i(e_i, p_i)$ has bounded deadline tardiness, then the total guaranteed long-term supply to container C_i will be proportional to the long-term supply of $m(C_i)$ fully available processors, which can be described by a set of $m(C_i)$ supply functions equal to Δ, plus that of a partially available processor with bandwidth $u_i = e_i/p_i$. We are left with characterizing the processor time that is available to C_i when the server task S_i is scheduled.

The amount of supply guaranteed to the server task S_i will depend on its parameters, e_i and p_i, the parameters of other SRT tasks, and the scheduling policy $\mathcal{Q}$ employed by the parent container H.

Lemma 1. *Let $\mathsf{A}(S_i, t_1, t_2, \mathcal{Q})$ be the total allocation of task S_i during the interval $[t_1, t_2)$ in the schedule $\mathcal{Q}$ and let Θ_i be the maximum deadline tardiness of task S_i's jobs in $\mathcal{Q}$. Then, the allocation $\mathsf{A}(S_i, 0, t, \mathcal{Q})$ satisfies the following.*

$$\mathsf{A}(S_i, 0, t, \mathcal{Q}) \le u_i \cdot t + e_i(1 - u_i) \tag{16}$$

$$\mathsf{A}(S_i, 0, t, \mathcal{Q}) \ge u_i \cdot t - u_i \cdot \Theta_i - e_i(1 - u_i) \tag{17}$$

The proof of this lemma is straightforward and thus is omitted due to space constraints. Instead, we illustrate (16) using an example.

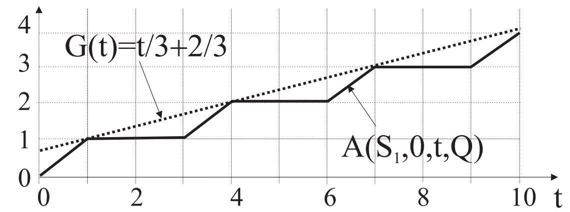


Figure 4. Server task allocation $\mathsf{A}(S_1, 0, t, \mathcal{Q})$ in Example 6 and its linear upper bound $G(t)$.

Example 6. Consider the schedule $\mathcal{Q}$ shown in Fig. 3(b). In this schedule, jobs of the server task $S_1(1, 3)$ execute in the intervals $[0, 1)$, $[3, 4)$, and $[6, 7)$. By time 1, S_1 has received one allocation unit, by time 4, its allocation is two units, and so on. The allocation $\mathsf{A}(S_1, 0, t, \mathcal{Q})$ is shown in Fig. 4 as a function of t. The figure also shows the upper bound (16), which is $G(t) \triangleq u_i \cdot t + e_i(1-u_i) = 1/3 \cdot t + 1(1-1/3) = 1/3 \cdot t + 2/3$. It is easy to see that $\mathsf{A}(S_1, 0, t, \mathcal{Q}) \leq G(t)$.

We now can find guarantees on the supplied processor time for server tasks for an arbitrary time interval.

Theorem 2. *Suppose that the scheduling algorithm used by the container H ensures a deadline tardiness bound of Θ_i for the server task $S_i(e_i, p_i)$. Then S_i is guaranteed at least $\phi_i^l(\Delta) = \max(0, u_i \cdot \Delta - 2e_i(1 - u_i) - u_i \cdot \Theta_i)$ time units during an interval of length Δ.*

Proof. Our goal is to bound the allocation of S_i during an interval $[t_1, t_2)$ by a function of the length of the interval $\Delta = t_2 - t_1$.

$$\begin{aligned}
&\mathsf{A}(S_i, t_1, t_2, \mathcal{Q}) \\
&= \mathsf{A}(S_i, 0, t_2, \mathcal{Q}) - \mathsf{A}(S_i, 0, t_1, \mathcal{Q}) \\
&\geq \{\text{by (16) and (17)}\} \\
&\quad u_i \cdot t_2 - u_i \cdot \Theta_i - e_i(1-u_i) - (u_i \cdot t_1 + e_i(1-u_i)) \\
&= u_i \cdot (t_2 - t_1) - 2e_i(1 - u_i) - u_i \cdot \Theta_i \\
&= u_i \cdot \Delta - 2e_i(1 - u_i) - u_i \cdot \Theta_i.
\end{aligned}$$

The allocation $\mathsf{A}(S_i, t_1, t_2, \mathcal{Q})$ cannot be less than zero, thus $\mathsf{A}(S_i, t_1, t_2, \mathcal{Q}) \geq \max(0, u_i \cdot \Delta - 2e_i(1-u_i) - u_i \cdot \Theta_i)$. $\square$

Corollary 2. *The supply to container C_i, as defined above, is described by $M(C_i) = \lceil w(C_i) \rceil$ availability functions in MP form, where $m(C_i) = \lfloor w(C_i) \rfloor$ supply functions satisfy $\beta_j^l(\Delta) = \Delta$ and at most one supply function satisfies $\beta_1^l(\Delta) = \phi_i^l(\Delta)$ as given by Theorem 2. The total supplied bandwidth for C_i is $w(C_i)$.*

Applying Corollaries 1 and 2 recursively, we can analyze the properties of a container hierarchy. In the case when there are no HRT tasks in the system, no utilization loss is incurred. However, the tardiness of SRT tasks may be higher as compared to a corresponding non-hierarchical approach, where all tasks are scheduled at the same level. This is the price for having temporal isolation among containers.

6. Experiments

We now present the results of experiments conducted to compare our container-aware scheduling scheme with conventional scheduling techniques. In these experiments, performance was compared using randomly-generated task sets, which have both HRT and SRT tasks.

Task generation procedure. We generated tasks for a single container by generating a set of HRT tasks τ_{hrt} and a set of SRT tasks τ_{srt}. Randomly-generated tasks were added to these sets while $U(\tau_{hrt}) \leq 1$ and $U(\tau_{srt}) \leq 8.5$, so that the size of the HRT component is small. Task utilizations were taken randomly from $[0, 0.15)$ for HRT tasks and from $[u_{min}, u_{max})$ for SRT tasks. We examined four SRT utilization ranges, as described later. Integral task periods were taken randomly from $[10, 50]$ for HRT tasks and from $[10, 100]$ for SRT tasks. Integral execution times were computed using periods and utilizations.

In order to gain intuition about the properties of a large multiprocessor platform running multiple isolated components, each generated container was replicated four times, giving a three-level hierarchy: a root container C_0, four next-level containers, and then the contained tasks. The i-th second-level container is denoted $C_{sys}^{[i]}$ and its contained tasks as $\tau_{hrt}^{[i]}$ and $\tau_{srt}^{[i]}$.

We compared our container-aware scheduling scheme (CA) with PEDF and a hybrid EDF-based scheme (HS), both of which are oblivious to containers. The HS scheme, which is described later in this section, is a naïve combination of PEDF and GEDF. PEDF was selected because it exhibits good timeliness, and HS was selected because it can satisfy the requirements of HRT and SRT tasks using relatively few processors. However, HS and PEDF do not provide any isolation among containers. In our experiments, we compared the tested schemes based on *the required number of processors* (RNP) and *deadline tardiness bound* (TB). Due to a lack of space, we did not consider any system overheads, and are unable to present results for other container hierarchies.

Defining RNP. Under PEDF, RNP was defined as the minimum number of processors required to partition all real-time tasks using the first-fit heuristic. Under PEDF, all tasks have zero tardiness.

Under HS, HRT and SRT tasks run on disjoint processor sets, with all HRT tasks scheduled together using PEDF with the first-fit heuristic, and all SRT tasks scheduled together using GEDF. RNP for the SRT tasks is thus $M_{soft} =$

$\lceil \sum_{i=1}^{4} U_{sum}(\tau_{srt}^{[i]}) \rceil$. Letting M_{hard} denote the HRT RNP, overall RNP under HS is simply $M_{hard} + M_{soft}$.

Under CA, we set container $C_{sys}^{[i]}$'s bandwidth as follows. Because $U(\tau_{hrt}^{[i]}) \leq 1$, the HRT tasks of each replica require one processor. According to our scheme, HRT tasks that belong to different replicas will always execute on different processors and meet their deadlines. Because the container $C_{sys}^{[i]}$ needs a dedicated processor for its HRT tasks, its bandwidth was set to

$$w(C_{sys}^{[i]}) = 1 + \begin{cases} \lfloor U(\tau_{srt}^{[i]}) \rfloor + \frac{1}{\pi_i} & \text{if } \lfloor U(\tau_{srt}^{[i]}) \rfloor \neq U(\tau_{srt}^{[i]}) \\ U(\tau_{srt}^{[i]}) & \text{otherwise,} \end{cases}$$

where π_i is the period of a server task $S_{srt}^{[i]}(1, \pi_i)$ defined so that $w(C_{sys}^{[i]}) \geq U(\tau_{srt}^{[i]})$, if $\pi_i > 1$. Choosing the container bandwidth this way instead of $w(C_{sys}^{[i]}) = 1 + U(\tau_{srt}^{[i]})$ may introduce utilization loss in addition to the loss in the encapsulated HRT container because the reserved bandwidth for SRT tasks may be greater than their total utilization. However, choosing the execution time of the server task $S_{srt}^{[i]}$ equal to one effectively reduces its maximum tardiness, and correspondingly, the supply blackout time, as (11) and Theorem 2 suggest. Overall, RNP for CA is simply the bandwidth of the root container C_0, $w(C_0) = \lceil \sum_{i=1}^{4} w(C_{sys}^{[i]}) \rceil$.

RNP results. Fig. 5(a) shows RNP results for PEDF, HS, and CA, for the cases when SRT tasks have low, medium, and high per-task utilizations, as given by the ranges $[0.01, 0.1)$, $[0.1, 0.5)$, and $[0.5, 1)$, respectively. We also examined the SRT utilization range $[0.6, 0.8)$ as well, as it is an extreme case where PEDF shows poor performance. For each utilization range, 100 task sets were generated and their RNP averaged. The figure also shows the average total system utilization, so that we can estimate the utilization overhead associated with each scheme.

As SRT per-task utilization increases, RNP for PEDF also increases because more processors are needed to bin-pack SRT tasks. The extreme case is the utilization range $[0.6, 0.8)$, where each SRT task requires a separate processor.

When SRT per-task utilizations are low, PEDF exhibits the best performance, since it can partition all tasks using a minimal number of processors. Under CA, the four replicas of the HRT task set require four processors, while under HS, all HRT tasks may be packed onto a smaller number of processors.

When SRT per-task utilizations are medium and high, the difference between HS and CA is minimal, with RNP for CA slightly higher, due to the utilization loss associated with the selection of container bandwidths. PEDF requires the largest number of processors because each SRT task with utilization exceeding 0.5 requires a separate processor.

Tardiness. Fig. 5(b) shows the average of the per-task-set tardiness bounds under HS and CA for the task set categories discussed above (under PEDF, tardiness is zero). For these two schemes, these tardiness bounds are comparable.

Overall, these experiments show that there is a price to be paid for temporal isolation among containers, in the form of more required processors (if HRT tasks are present) or higher tardiness. However, in our proposed scheme, this price is reasonable, when considering the performance of schemes that ensure no isolation.

7. Conclusion

We have presented a multiprocessor bandwidth-reservation scheme for hierarchically organized real-time containers. Under this scheme each real-time container can reserve any fraction of processor time (even the capacity of several processors) to schedule its children. The presented scheme provides temporal isolation among containers so that each container can be analyzed separately.

Our scheme is novel in that soft real-time components incur no utilization loss. This stands in sharp contrast to hierarchical schemes for hard (only) real-time systems, where the loss per level can be so significant, arbitrarily deep hierarchies simply become untenable.

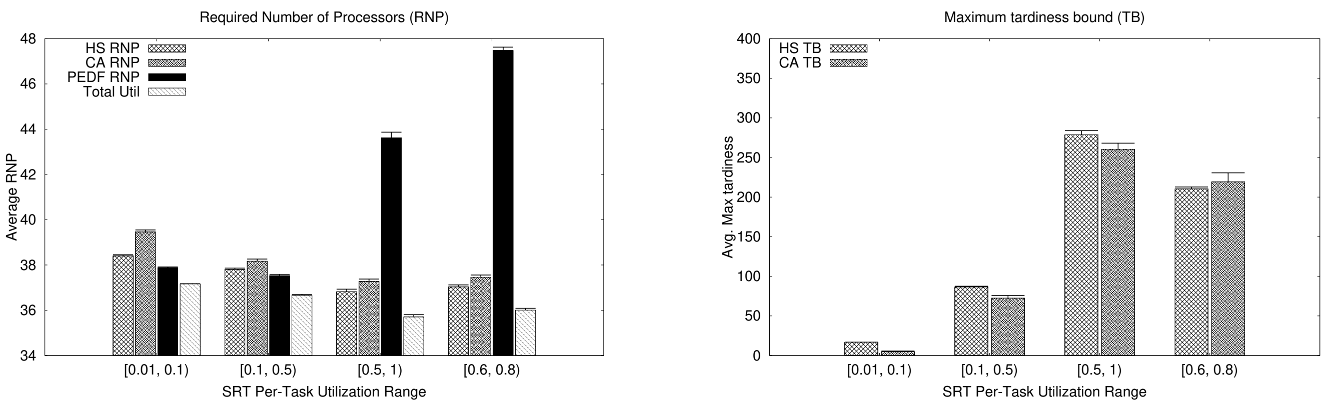
Several interesting avenues for further work exist. The most important open problem is to enable dynamic container creation and the joining/leaving of tasks. Also of importance is the inclusion of support for synchronization. It would also be interesting to investigate other global scheduling algorithms such as Pfair algorithms to see whether a more accurate analysis can be established for them. Finally, the new scheduling policy needs to be implemented on a real multiprocessor platform so that overheads associated with the hierarchical nature of the system could be determined. Given that work on Linux containers partially motivated our research, a Linux-based system such as LITMUS$^{\mathrm{RT}}$ [15], where GEDF is implemented along with other global scheduling algorithms, would be desirable to use in such an effort.

References

[1] J. Anderson, J. Calandrino, and U. Devi. Real-time scheduling on multicore platforms. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 179–190, April 2006.

[2] S. Baruah. Techniques for multiprocessor global schedulability analysis. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 119–128, December 2007.

[3] S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of sporadic task systems. In *Proceedings of the*



(a) (b)

Figure 5. (a) Required number of processors and (b) maximum tardiness bounds for randomly generated task sets (with 95% confidence intervals).

IEEE Real-Time Systems Symposium, pages 321–329, December 2005.

[4] 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, September 2002.

[5] S. Baruah and G. Lipari. A multiprocessor implementation of the total bandwidth server. In *Proceedings of 18-th International Parallel and Distributed Processing Symposium*, page 40, April 2004.

[6] B. Brandenburg and J. Anderson. Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 61–70, July 2007.

[7] S. Chakraborty and L. Thiele. A new task model for streaming applications and its schedulability analysis. In *Proceedings of the IEEE Design Automation and Test in Europe (DATE)*, pages 486–491, March 2005.

[8] U. Devi and J. Anderson. Tardiness bounds for global EDF scheduling on a multiprocessor. *Real-Time Systems*, 38(2):133–189, February 2008.

[9] A. Easwaran, M. Anand, and I. Lee. Compositional analysis framework using EDP resource models. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 129–138, 2007.

[10] M. Eriksson and S. Palmroos. Comparative study of containment strategies in solaris and security enhanced Linux. June 2007. http://opensolaris.org/os/community/security/news/20070601-thesis-bs-eriksson-palmroos.pdf.

[11] P. Holman and J. Anderson. Group-based Pfair scheduling. *Real-Time Systems*, 32(1-2):125–168, February 2006.

[12] H. Leontyev and J. Anderson. Tardiness bounds for FIFO scheduling on multiprocessors. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, July 2007. 71-80.

[13] P. Lessard. Linux process containment: A practical look at chroot and user mode Linux. 2003. http://www.sans.org/reading_room/whitepapers/linux/1073.php.

[14] Linux vserver documentation. September 2007. http://linux-vserver.org/Documentation.

[15] LITMUS$^{\text{RT}}$ homepage. http://www.cs.unc.edu/~anderson/litmus-rt.

[16] J.W.S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[17] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 294–303, December 1999.

[18] 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, March 2006.

[19] 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, July 2006.

[20] I. Shin, A. Easwaran, and I. Lee. Compositional analysis of hierarchical multiprocessor scheduling frameworks. Private communication, 2007.

[21] 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, June 2002.