Parallel Task Scheduling on Multicore Platforms *

James H. Anderson and John M. Calandrino
Department of Computer Science, The University of North Carolina at Chapel Hill

Abstract

We propose a scheduling method for real-time systems implemented on multicore platforms that encourages certain groups of tasks to be scheduled together while ensuring real-time constraints. This method can be applied to encourage tasks that share a common working set to be executed in parallel, which makes more effective use of shared caches.

1 Introduction

Multicore architectures, which include several processors on a single chip, are being widely touted as a solution to thermal and power problems currently limiting single-core designs. In most proposed multicore platforms, different cores share either on- or off-chip caches. In this paper, we are concerned with the efficient utilization of such caches. To reasonably constrain the discussion, we henceforth limit attention to the multicore architecture shown in Fig. 1, wherein all cores are symmetric and share a chip-wide L2 cache, and each core supports one hardware thread. To effectively exploit the available parallelism on these platforms, shared caches must not become performance bottlenecks. In prior work pertaining to throughput-oriented systems, Fedorova *et al.* [6] noted that L2 misses affect performance to a much greater extent than L1 misses or pipeline conflicts. They showed that L2 contention can be reduced, and throughput improved, by discouraging threads that generate significant memory-to-L2 traffic from being co-scheduled.

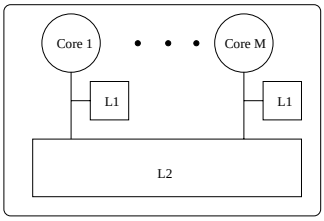

Figure 1: Multicore architecture.

The problem. In previous work [1], we addressed the issue of whether, in real-time systems, tasks could be discouraged from being co-scheduled *while ensuring real-time constraints*. This paper addresses the opposite issue of whether tasks can be *encouraged* to be co-scheduled. For example, we may wish to co-schedule a set of tasks that share a common working set. Fedorova *et al.* claimed that, for throughput-oriented applications, co-scheduling tasks that share a common working set would provide little performance benefit. We question the validity of this claim for real-time systems. To simply the problem, we assume that all of the tasks that we want to co-schedule have the same *utilization*, or *weight*. This restriction also minimizes the duration of time that a parallel-accessed working set (WS) may create cache traffic. For example, with four tasks of weight $3/4$, $1/4$, $1/2$, and $1/2$, the WS is accessed 75% of the time, as opposed to 50% if the task weights were each $1/2$ and perfect parallelism ensured.

Related work. In work on (non-multicore) systems that support *simultaneous multithreading* (*SMT*), prior work on *symbiotic scheduling* is of relevance to our work [7, 8, 10]. In symbiotic scheduling, the goal is to maximize the overall "symbiosis factor," which is a measure that indicates how well various thread groupings perform when co-scheduled. To the best of our knowledge, no analytical results concerning real-time constraints have been obtained in work on symbiotic scheduling. Additionally, prior work such as [3] has addressed the issue of scheduling tasks that require more than one processor. This could be seen as being equivalent to the problem of co-scheduling a set of equal-weight tasks. However, this prior work places restrictions on scheduling, such as requiring that certain tasks be pinned to certain processors, and provides few analytical results.

The need to encourage certain tasks to be co-scheduled fundamentally distinguishes the problem at hand from other real-time multiprocessor scheduling problems considered previously [5]. Providing support for tasks that should be encouraged to execute together is harder than discouraging tasks from executing together. We claim in Sec. 2 that the general problem of optimizing for parallelism while respecting real-time constraints is NP-hard in the strong sense. Because of this limitation, we have instead focused our efforts on minimizing a factor we call *spread*: if a task grouping has a spread of k, then the i^{th} quantum of computation of each task must be scheduled within an interval $[t, t + k)$ for some t (treating each quantum as a "time unit"). Perfect parallelism corresponds to a spread of one. We have devised a "spread-cognizant" scheduling method and applied it to both the PD^2 Pfair scheduling algorithm and global EDF. This method is capable of decreasing both average and maximum spread. Additionally, we can *guarantee* a maximum spread for a task set,

*Work supported by NSF grants CCR 0309825 and CCR 0408996.

based on the maximum weight of any task in the set, when using either PD^2 or global EDF as the scheduling algorithm.

The rest of this paper is organized as follows. We discuss impossibility and intractability results for parallelism in Sec. 2. In Sec. 3, we present a brief overview of Pfair scheduling. Sec. 4 describes our spread-cognizant scheduling method, and cases where we can formally bound the spread of groups of tasks to be co-scheduled. In Sec. 5, we discuss the performance of our method in terms of both spread and L2 cache-miss ratios on a simulated multicore platform. Finally, we consider some directions for future work in Sec. 6.

2 The Task Co-scheduling Problem

The essence of the problem at hand is to provide support for *parallelism*: when one of a group of tasks is scheduled, we would like *all* tasks in the group to be scheduled. Unfortunately, achieving perfect parallelism for all task groups is not always possible. For example, with one task of weight $3/4$ and two tasks of weight $1/2$, we cannot co-schedule all tasks of weight $1/2$ on a two-processor system unless at least one task misses a deadline. Either the tasks of weight $1/2$ keep both processors busy half of the time, and thus the task of weight $3/4$ cannot meet its deadlines, or the $3/4$-wt. task meets its deadlines and both processors are not simultaneously available for enough time to co-schedule the other tasks. Note that, without the co-scheduling requirement, we can schedule all tasks easily: allocate one processor to the $3/4$-wt. task and the other to the two $1/2$-wt. tasks.

Additionally, we have shown that the general problem of optimizing for parallelism while respecting real-time constraints is NP-hard in the strong sense, by a transformation from 3-PARTITION. Therefore, we must instead rely on suboptimal heuristics. The rest of this paper is dedicated to the design of such a heuristic, described further in Sec. 4.

3 Background on Pfair Scheduling

We briefly digress to introduce Pfair scheduling [4, 11]. Let τ denote a set of periodic tasks to be scheduled on M processors (or cores). Each task $T \in \tau$ has a rational *weight* $T.w \in (0, 1]$ equal to its utilization. Conceptually, $T.w$ is the rate at which T should execute, relative to the speed of a single processor. Pfair scheduling algorithms allocate processor time in discrete time units called *quanta*. The time interval $[t, t + 1)$, where $t \geq 0$, is called *slot* t. In each slot, each processor (task) can be assigned to at most one task (processor). Task migration is allowed.

Under Pfair scheduling, each task T is sub-divided into a sequence of quantum-length *subtasks*, denoted $T_1, T_2, \ldots$. Each subtask must be scheduled in a certain time-slot interval called its *window*, the end of which is its *deadline*. A task's windowing is defined so that its allocation approximates that

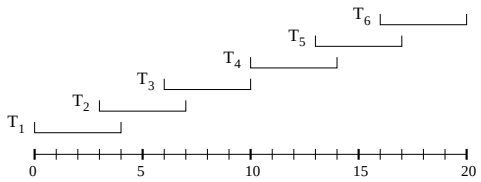

Figure 2: Windowing for a task with weight $T.w = 3/10$ under Pfair scheduling over the interval [0,20]. Note that three (six) subtasks have deadlines by time 10 (20). Thus, if each subtask meets its deadline, then T's allocation up to these times is the same as in an ideal system ($\frac{3}{10} \times 10$ and $\frac{3}{10} \times 20$, respectively).

of an ideal (fluid) system that allocates $T.w \cdot L$ units of processor time to each task T in any interval of length L. Fig. 2 shows an example.

Pfair scheduling algorithms. Pfair scheduling algorithms schedule tasks by scheduling their subtasks on an earliest-deadline-first basis. Tie-breaking rules are used in case two subtasks have the same deadline. The most efficient optimal algorithm known is PD^2 [2, 11], which uses two tie-breaking rules. The term "optimal" means that these algorithms are capable of ensuring the timing requirements of all tasks as long as the system's total utilization, $\sum_{T \in \tau} T.w$, does not exceed M, the number of available processors.

Tardiness bounds. If a scheduling algorithm cannot ensure all timing requirements, then it may still be possible to show that deadline tardiness is always bounded by some number of quanta. A *tardiness bound* of B means that a subtask will be executed at most B quanta after its deadline.

Early-release scheduling. In the *early-release (ER) task model* [2], a subtask may be released early, *i.e.* be eligible to execute before its window, while still ensuring the timing requirements of all tasks. The decision on whether or not to release a subtask early, and how early to release it, can be arbitrary, except that all subtasks of a task must execute in order.

4 Spread-Cognizant Scheduling

Our approach for minimizing spread while meeting real-time constraints is based upon three observations:

- Global scheduling algorithms, which use a single run queue, are more naturally suited to minimizing spread than partitioning approaches. This is particularly the case when using deadline-based scheduling methods. This is because "work" submitted at the same time by same-weight tasks will occupy consecutive slots in the scheduler's run queue, and thus, such work will be scheduled in close proximity over time, unless disrupted by later-arriving, higher-priority work. Based on this observation, we limit our attention to global, deadline-based

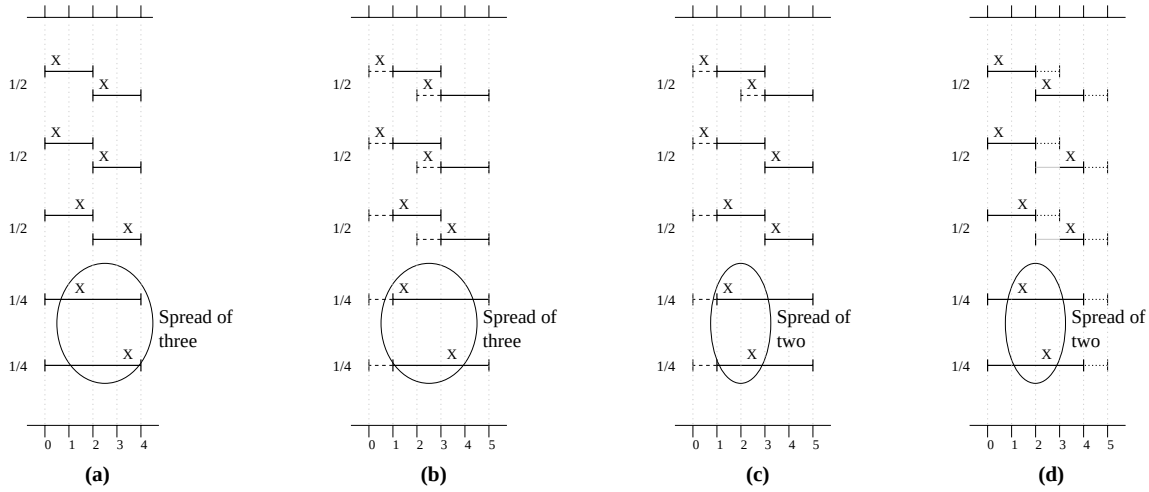

Figure 3: A two-processor Pfair schedule of a set of five tasks (three of weight 1/2, and two of weight 1/4) with **(a)** no early releasing; **(b)** early releasing by one quantum and all windows right-shifted by one quantum; **(c)** similarly-shifted windows, but with selective early releasing; and **(d)** no shifting or early releasing, but subtasks are considered optional within the first quantum after their release, and deadlines can be missed by one quantum.

scheduling approaches. Two such approaches we consider are the PD^2 Pfair scheduling algorithm and the global earliest-deadline-first (EDF) algorithm. In Pfair scheduling, each unit of work submitted by a task is a quantum-length *subtask*, as noted earlier, while under EDF, the work submitted is in the form of *jobs* of arbitrary (but bounded) length.

- In all global, deadline-based scheduling methods known to us, the ability to meet timing constraints is not compromised if subtasks or jobs (as the case may be) are "early released," *i.e.*, allowed to become eligible for execution "early."

- When a subtask or job is early released, it is completely *optional* as to whether the scheduler considers it to be available for execution. We can exploit this fact to minimize disruptions to task groups caused by higher-priority work.

As an example, consider the Pfair schedules in Fig. 3, where both tasks of weight $1/4$ are placed in their own task group. Inset (a) shows a schedule without early releasing. In inset (b), all subtask windows are shifted right by one quantum, and all tasks are early released by one quantum, producing the same schedule as in (a). Dashed lines before each window indicate how early each subtask could be scheduled. We refer to a schedule in which all subtask windows are right-shifted by k quanta and all subtasks are early released by k quanta as a *k-shifted schedule*. In inset (b), $k = 1$. Both schedules result in a spread of three for the $1/4$-wt. task group. In inset (c), we show that selectively allowing early releasing can reduce spread to two. Alternately, instead of shifting the schedule and early releasing subtasks, as in insets (b) and (c), we can instead make it completely optional whether to schedule subtasks for the first k quanta after their release, and allow jobs to miss their deadlines by up to k quanta, as shown in inset (d). Here, the dotted lines after each window indicate by how much each deadline could be missed (though no misses occur here). Note that there are "intermediate" cases between an unshifted and a k-shifted schedule. For example, if $k = 4$ is required by our method for a particular task set, then we could choose instead to create a 3-shifted schedule and allow jobs to early release by only three quanta, but also make it optional whether to schedule jobs during the first quantum after their actual release. In this case, deadlines could be missed by at most one quantum. In general, if subtasks can be early released to the extent we require, then no deadlines will be missed; otherwise, deadlines may be missed, but by bounded amounts only.

The experiments in Sec. 5 show that we usually get spread reductions when applying our method. Additionally, Theorem 1 states a condition under which a spread of X can be guaranteed when scheduling using PD^2. X is defined as follows, where $W_{\max} = \max_{T \in \tau} wt(T)$.

$$X = \begin{cases} 3, & \textbf{if } W_{\max} \leq 1/3 \\ 4, & \textbf{if } 1/3 < W_{\max} \leq 1/2 \\ 2 \times \lceil \frac{1}{1-W_{\max}} \rceil - 1, & \textbf{if } W_{\max} > 1/2 \end{cases} \quad (1)$$

Theorem 1 *If PD^2 is modified as described above, and subtasks are early-release eligible $X - 1$ quanta before their actual release times, then the spread of any group of tasks that we wish to co-schedule is no greater than X as defined in* (1).

Spread for EDF. Theorem 2 states a similar spread result for EDF. In EDF scheduling, jobs of periodic tasks are prioritized by increasing order of their deadlines.

Algorithm	Wt. Constr.	X	ER	Spread Grp. Size = 2 Min	Avg	Max	Grp. Size = 3 Min	Avg	Max	Grp. Size = 4 Min	Avg	Max
Reg. Pfair	$(0, 1/3]$	N/A	0	1	1.35	**41**	1	1.66	**40**	1	1.99	**41**
Mod. Pfair	$(0, 1/3]$	3	2	1	1.27	**2**	1	1.52	**2**	1	1.77	**3**
Reg. Pfair	$(0, 1/2]$	N/A	0	1	1.40	**37**	1	1.78	**41**	1	2.18	**37**
Mod. Pfair	$(0, 1/2]$	4	3	1	1.28	**2**	1	1.53	**2**	1	1.77	**3**
Reg. Pfair	$(0, 3/4]$	N/A	0	1	1.39	**25**	1	1.83	**33**	1	2.29	**41**
Mod. Pfair	$(0, 3/4]$	7	6	1	1.29	**2**	1	1.57	**2**	1	1.81	**3**

(a)

Algorithm	e_{max}	X	ER	Spread Grp. Size = 2 Min	Avg	Max	Grp. Size = 3 Min	Avg	Max	Grp. Size = 4 Min	Avg	Max
Reg. EDF	1	N/A	0	1	1.96	**41**	1	2.28	**42**	1	2.54	**45**
Mod. EDF	1	3	2	1	1.34	**2**	1	1.47	**2**	1	1.62	**2**

(b)

Table 1: Spread for **(a)** Pfair and **(b)** EDF algorithms. Each different constraint represents 50,000 task sets.

Theorem 2 *Consider a task set τ for which tardiness is at most Δ under EDF, and let e_{max} denote the largest job execution cost in τ. Suppose that EDF is modified as described above for PD^2, but instead jobs are allowed to become early-release eligible up to $2 \cdot e_{max}$ quanta before their actual release times. If $T.p \geq T.e + 1 + \Delta$ for each task $T \in \tau$, then the spread of any task group is at most $2 \cdot e_{max} + 1$.*

5 Experimental Results

In this section, we assess the efficacy of our method in reducing spread and achieving better L2 cache performance.

Spread reduction experiments. First, we randomly generated 50,000 task sets in several categories, and simulated the scheduling of these task sets on a four-processor system with EDF and PD^2. For each task set with which we experimented, we first allowed no early-releasing and did not shift the schedule, and then allowed early-releasing and shifted by $X - 1$ quanta, as specified earlier for both PD^2 and EDF. For PD^2, an upper bound on task weights was enforced, during task set generation, of $1/3$, $1/2$, or $3/4$, depending on the experiment. For EDF, all tasks had an execution cost of one. Task periods varied from two (or three, if tasks could not have a weight of $1/2$) to 50. All task sets fully utilized all four processors, and the task groups varied in size from one (*i.e.*, a lone task in its own task "group") to four (the total number of processors). These constraints were reasonable as they included task sets with a wide variety of task weights, including those with large periods (*e.g.*, 50). The only types of tasks not included were tasks with weight greater than $3/4$ in PD^2 or execution cost over one in EDF. All simulations were run for the length of the task-set hyperperiod. Results are shown in Table 1.

Results. Our method generates low spreads in all cases (near two quanta), thus appearing to perform considerably better in practice than might be expected from the analytical upper bounds on spread for PD^2 stated in Sec. 4. For the task sets used in the global EDF experiments, $T.p \geq T.e + 1 + \Delta$ was *not* explicitly ensured for each task T in the EDF task sets. However, the spread results are still impressive and within the analytical bound for EDF stated in Sec. 4. Note that our method decreases average spread and *always prevents extremely high spreads*, as shown in the boldface columns of Table 1.

L2 cache performance. We next demonstrate the effectiveness of our method at improving L2 cache performance by simulating the scheduling and execution of task sets with the SESC Simulator [9], which is capable of simulating a variety of multicore architectures. We chose to use a simulator so that we could experiment with more cores than commonly available today. The simulated architecture consists of four cores, each with dedicated 16K L1 data (4-way set assoc.) and instruction (2-way set assoc.) caches with random and LRU replacement policies, respectively, and a shared 8-way set assoc. on-chip L2 cache with an LRU replacement policy (size varies per experiment). Each cache has a 64-byte line size.

All tasks in the same group access the same large "working set" region of memory. Tasks make one pass over a region of their working set per quantum of execution, and therefore only tasks in the same group can utilize the cache by reusing blocks already brought in by other tasks in that group. To encourage such reuse, all tasks in the same group access the same region of memory for a given quantum of computation, and each task starts accessing the region in a different location, wrapping if necessary. If all tasks started accessing memory in the same location, all tasks would proceed in a "lock step" manner while waiting for blocks to be loaded into the cache from main memory, resulting in virtually no benefit from the cache. By starting in different locations, tasks in the group can better reuse what remains in the cache later in the quantum.

The region of memory accessed each quantum is determined by a "sliding window" that moves by 15,000 cache

blocks over the "working set" region of memory shared by all tasks in the group. This memory access pattern is intended to account for the worst-case memory performance of tasks and keeps groups of tasks synchronized with one another. Due to this memory access pattern, tasks access approximately 960K of memory per quantum. Thus, the sliding window allows for some level of cache reuse that decreases and eventually drops off entirely with increasing spread, due to capacity and conflict misses.

One application with the potential to exhibit a memory access pattern similar to the one described might be parallel motion compensation search, which is the most compute-intensive part of MPEG-2 video encoding. In such an application, some number of tasks would access the same region of memory during the search. However, each task would start accessing the region in a different location. Such an application might encode a video stream in real time on a frame-by-frame basis, and therefore would require (soft) real-time guarantees. Additionally, there would clearly be some benefit to co-scheduling tasks that are encoding the same frame (during the same quantum of computation).

Hand-crafted task sets. We created several hand-crafted task sets to demonstrate the effectiveness of our method at reducing L2 cache-miss rates. The hand-crafted task sets are listed in Table 2. We allowed early-releasing and shifted the schedule by the indicated number of quanta when applicable. Each task set was run for 20 quanta (assuming a 0.75-ms quantum length) on an architecture with the specified number of cores and the indicated L2 cache size. Table 3 shows for each case the L2 cache-miss rates that were observed.

While the SESC Simulator is very accurate, it comes at the cost of being quite slow. Therefore, longer and more detailed results could not be obtained because of the length of time it took the simulations to run. Additionally, space constraints were also a limiting factor in the amount of experimental data we could present. We hope to present more extensive experiments in a future paper.

The L2 cache-miss rates given in Table 3 show that our method can result in substantially better performance. Note that the opportunities for cache reuse are limited by our memory access pattern, and therefore all miss ratios are quite high. However, our method shows an overall improvement with these task sets. In task sets BASIC, LONGER_BASIC, and ONE_PROC, one task group can benefit from the cache, and tasks in that group have relatively low weight. In order to see a significant overall benefit, the actual performance of tasks in that group must have dramatically improved, though the SESC Simulator gave us no way of measuring the cache performance of specific tasks. Additionally, because an L2 miss incurs a time penalty roughly two orders of magnitude greater than a hit, *a miss-rate difference can correspond to a significant difference in performance*, as seen in [1].

Name	No. Tasks	Task Properties	ER	No. Cores	L2 Size
BASIC	5	2 of Wt. 1/4 (same group) 3 of Wt. 1/2 (indep.)	1	2	2048K
LONGER_BASIC	5	2 of Wt. 1/10 (same group) 3 of Wt. 3/5 (indep.)	1	2	2048K
ONE_PROC	3	2 of Wt. 1/4 (same group) 1 of Wt. 1/2 (indep.)	1	1	1024K
MAX_PARA	3	16 of Wt. 1/4 (4 groups of 4 tasks)	2	4	2048K
NO_PARA	5	16 of Wt. 1/4 (indep.)	2	4	2048K

Table 2: Properties of hand-crafted task sets.

Name	Reg. Pfair	Mod. Pfair	Reg. EDF	Mod. EDF
BASIC	79.57%	63.79%	79.18%	63.74%
LONGER_BASIC	60.54%	55.36%	40.27%	34.89%
ONE_PROC	80.93%	52.53%	80.92%	52.60%
MAX_PARA	23.43%	23.35%	23.45%	23.46%
NO_PARA	79.02%	78.73%	78.72%	79.02%

Table 3: L2 cache miss ratios for hand-crafted task sets.

BASIC and LONGER_BASIC consist of three independent tasks with large weight, and two tasks of smaller weight in the same task group. A spread of at least three is achieved for the task group of each set without our method, and thus little opportunity for cache reuse exists. With our method, spread is reduced to two, and the cache is better utilized, resulting in decreased miss rates. Note that LONGER_BASIC shows less of an overall improvement, as the tasks in the group are of lower weight and therefore there is a smaller improvement in overall cache performance.

ONE_PROC consists of one independent task with weight $1/2$, and two tasks of weight $1/4$ in the same task group. As there is only one processor, these two tasks cannot be co-scheduled, but we can achieve a spread of two with our method. This allows one task in the group to directly reuse data brought in from the other task during the preceding quantum, and we therefore see the most impressive benefit from reducing spread in this case.

MAX_PARA and NO_PARA demonstrate the dramatic cache benefits that can be achieved when tasks that share memory regions achieve a spread of one. Every quantum with set MAX_PARA, each block of memory brought into the cache is reused three times, resulting in approximately a 25% cache-miss ratio in all cases. Alternately, NO_PARA provides no opportunity for reuse, and we see very high cache-miss ratios as a result.

We emphasize that these example task sets demonstrate that our method *can* lead to lower L2 miss rates. However, there will not *always* be a benefit, and this is why it is difficult to demonstrate a benefit for some number of randomly-generated task sets, as was done in the spread reduction experiments presented earlier. Determining whether our method will improve cache performance for a particular application is difficult because it depends on many factors related to how memory is accessed, such as the percentage of reads and writes, opportunities for reuse (*i.e.*, reuse distances), and other aspects of the overall memory access pattern. For many appli-

cations, these factors could vary dramatically over each quantum of computation, which further complicates the issue. We claim, however, that our method will not have a dramatic *negative* effect on cache-miss rates.

6 Future Work

There are many directions for future work. First, we want to further generalize these methods to other scheduling algorithms, including non-preemptive policies. Second, we want to combine this scheduling method with the methods in [1] so that both the "encouragement" and "discouragement" of co-scheduling can be supported in the same system. Third, our preliminary work has only considered independent tasks that do not interact. Given that the notion of minimizing spread might produce a significant benefit to groups of tasks that require some level of coordination, we wish to explore support for critical sections and precedence constraints that incorporate our spread-cognizant scheduling policies. Fourth, we currently assume that all tasks to be co-scheduled have the same weight; we would like to remove this constraint. Finally, we want to find other applications that showcase our method, and ultimately implement and test those applications on a real multicore system.

References

[1] J. Anderson, J. M. Calandrino, and U. Devi. Real-time scheduling on multicore platforms. To appear in *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2006.

[2] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. *Journal of Computer and System Sciences*, 68(1):157–204, 2004.

[3] E. Bampis, M. Caramia, J. Fiala, A. V. Fishkin, and A. Iovanella. Scheduling of independent dedicated multiprocessor tasks. *13th Annual International Symposium on Algorithms and Computation*, Nov. 2002.

[4] S. Baruah, N. Cohen, C.G. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[5] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In Joseph Y. Leung, editor, *Handbook on Scheduling Algorithms, Methods, and Models*, pages 30.1–30.19. Chapman Hall/CRC, Boca Raton, Florida, 2004.

[6] A. Fedorova, M. Seltzer, C. Small, and Daniel Nussbaum. Throughput-oriented scheduling on chip multithreading systems. Technical Report TR-17-04, Division of Engineering and Applied Sciences, Harvard University, August 2004.

[7] R. Jain, C. Hughs, and S Adve. Soft real-time scheduling on simultaneous multithreaded processors. In *Proceedings of the 23rd IEEE Real-time Systems Symposium*, pages 134–145. IEEE, December 2002.

[8] S. Parekh, S. Eggers, H. Levy, and J. Lo. Thread-sensitive scheduling for SMT processors. http://www.cs.washington.edu/ research/smt/.

[9] J. Renau. SESC website. http://sesc.sourceforge.net.

[10] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic job scheduling with priorities for a simultaneous multithreading processor. In *Proceedings of SIGMETRICS 2002*. ACM, 2002.

[11] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 189–198, May 2002.