The Case for Fair Multiprocessor Scheduling *

Anand Srinivasan, Philip Holman, James H. Anderson, and Sanjoy Baruah
Department of Computer Science
University of North Carolina at Chapel Hill
E-mail: {anands,holman,anderson,baruah}@cs.unc.edu

November 2002

Abstract

Partitioning and global scheduling are two approaches for scheduling real-time tasks on multiprocessors. Though partitioning is sub-optimal, it has traditionally been preferred; this is mainly due to the fact that well-understood uniprocessor scheduling algorithms can be used on each processor. In recent years, global-scheduling algorithms based on the concept of "proportionate fairness" (Pfairness) have received considerable attention. Pfair algorithms are of interest because they are currently the only known method for optimally scheduling periodic, sporadic, and "rate-based" task systems on multiprocessors. In addition, there has been growing practical interest in scheduling with fairness guarantees. However, the frequency of context switching and migration in Pfair-scheduled systems has led to some questions concerning the practicality of Pfair scheduling.

In this paper, we investigate this issue by comparing the PD^2 Pfair algorithm to the EDF-FF partitioning scheme, which uses "first fit" (FF) as a partitioning heuristic and the earliest-deadline-first (EDF) algorithm for per-processor scheduling. We present experimental results that show that PD^2 is competitive with, and in some cases outperforms, EDF-FF. These results suggest that Pfair scheduling is a viable alternative to partitioning. Furthermore, as discussed herein, Pfair scheduling provides many additional benefits, such as simple and efficient synchronization, temporal isolation, fault tolerance, and support for dynamic tasks.

Keywords: Fair scheduling, multiprocessors, partitioning, Pfairness, real-time systems.

*Work supported by NSF grants CCR 9972211, CCR 9988327, ITR 0082866, and CCR 0204312.

1 Introduction

Multiprocessor scheduling techniques in real-time systems fall into two general categories: *partitioning* and *global scheduling*. Under partitioning, each processor schedules tasks independently from a local ready queue. Each task is assigned to a particular processor and is only scheduled on that processor. In contrast, all ready tasks are stored in a single queue under global scheduling. A single system-wide priority space is assumed; the highest-priority task is selected to execute whenever the scheduler is invoked, regardless of which processor is being scheduled. In this paper, algorithms from both categories are considered and analyzed. Before summarizing our contributions, we first describe the advantages and disadvantages of both approaches in more detail.

Partitioning. Presently, partitioning is the favored approach. This is largely because partitioning has proved to be both efficient and reasonably effective when using popular uniprocessor scheduling algorithms, such as the earliest-deadline-first (EDF) and rate-monotonic (RM) algorithms [26]. Producing a competitive global scheduler, based on such well-understood uniprocessor algorithms, has proved to be a daunting task. In fact, Dhall and Liu have shown that global scheduling using either EDF or RM can result in arbitrarily-low processor utilization in multiprocessor systems [13].

Partitioning, regardless of the scheduling algorithm used, has two primary flaws. First, it is inherently suboptimal when scheduling periodic tasks.[1] A well-known example of this is a two-processor system that contains three synchronous[2] periodic tasks, each with an execution cost of 2 and a period of 3. Completing each job before the release of its successor for all tasks in such a system is impossible without migration. Hence, this task set is not schedulable under the partitioning approach.

Second, the assignment of tasks to processors is a bin-packing problem, which is NP-hard in the strong sense. Hence, optimal task assignments cannot be obtained online due to the run-time overhead involved. Online partitioning is typically done using heuristics, which may be unable to schedule task systems that are schedulable using offline partitioning algorithms.

Partitioning may introduce other problems as well. For example, in dynamic systems in which tasks may *join* and *leave*, partitioning is problematic because the arrival of a new task may necessitate a re-partitioning of the entire system, which will likely result in unacceptable overhead. This overhead will almost certainly be higher if tasks share resources. In fact, resource-sharing protocols have yet to be developed for partitioning with EDF. Though protocols have been proposed for partitioning with RM, the total utilization bound under RM can be as low as 41% *without* resource sharing [30].

Pfair scheduling. In recent years, much research has been done on global multiprocessor scheduling algorithms that ensure fairness [2, 3, 4, 6, 9, 10, 16, 17, 18, 29, 38, 39]. *Proportionate-fair* (Pfair) scheduling, proposed by Baruah *et al.* [5], is presently the only known optimal method for scheduling recurrent real-time tasks on a multiprocessor system. Under Pfair scheduling, each task is assigned a *weight* that specifies the rate at which that task should execute: a task with weight w would ideally receive $w \cdot L$ units of processor time over any interval of length L. Under Pfair scheduling, tasks are scheduled according to a fixed-size allocation quantum so that deviation from an ideal allocation is strictly bounded. Currently, three optimal Pfair scheduling algorithms are known: PF [5], PD [6], and PD^2 [2]. Of these algorithms, PD^2 is the most recently developed and the most efficient.

The primary advantage of Pfair scheduling over partitioning is the ability to schedule any feasible periodic, sporadic, or rate-based task system[3] [2, 4, 39]. Hence, Pfair scheduling algorithms can

[1] A *periodic* task represents an infinite sequence of identical jobs with evenly-spaced releases. The fixed delay that separates consecutive job releases of a task is referred to as the task's *period*.

[2] The first job of each task is released at time 0.

[3] Whereas a periodic task's period specifies an exact separation between job releases, a *sporadic* task's period specifies

seamlessly handle dynamic events, such as tasks leaving and joining a system. Furthermore, fair multiprocessor scheduling algorithms are becoming more popular due to the proliferation of web and multimedia applications. For instance, Ensim Corp., an Internet service provider, has deployed fair multiprocessor scheduling algorithms in its product line [22].

The main disadvantage of Pfair scheduling is degraded *processor affinity*. Processor affinity refers to the tendency of tasks to execute faster when repeatedly scheduled on the same processor. This tendency is usually the result of per-processor first-level caching. Preemptions and migrations, both of which tend to occur frequently under Pfair scheduling, limit the effectiveness of these first-level caches and can lead to increased execution times due to cache misses. On the other hand, under partitioning with EDF, there is no migration and the number of preemptions on a processor is bounded by the number of jobs on that processor (assuming independent tasks).

Current technological trends. It is useful to consider how various technological trends may impact the viability of the scheduling approaches considered above. Processor speeds have been increasing rapidly in recent years. As processors become faster, job execution times decrease. This typically leads to smaller task utilizations, because task periods are usually based upon external factors (such as human perception limits) rather than on processor speeds. In addition, an increase in processor speed can only reduce scheduling and context-switching costs. When both task utilizations and these overhead factors are low, the allocation of processor time almost becomes a non-problem, and the importance of efficiently managing other shared resources (such as critical sections) increases.

Despite these trends, systems will continue to exist in which processor time must be carefully allocated. This is especially true of embedded systems, which often use older and slower hardware to reduce costs. It is also true of emerging applications such as immersive-reality systems, which can be quite complex, and require compute-intensive tasks with high utilizations. High-utilization tasks can also arise under hierarchical scheduling approaches in which many tasks are bundled together and scheduled as a single task [16, 29].

As mentioned earlier, Pfair scheduling algorithms tend to migrate tasks frequently. As such, Pfair algorithms may not be a good choice for loosely-coupled systems. In this paper, we consider only tightly-coupled shared-memory systems. In such systems, the degraded performance caused by a migration (beyond that caused by a preemption without migration) is almost entirely due to the fact that, after the migration, references to memory locations accessed previously will not be serviced in-cache. In a tightly-coupled system, the resulting overhead should be small. In 1991, Mogul and Borg [28] showed that refilling a cache after a preemption can take from 10 μs to 400 μs. Though modern caches are larger, they are also much faster. Thus, this delay is likely much shorter now. In embedded systems, caches, if present, will typically be slower but also much smaller. Hence, cache-miss penalties in such systems will often be small as well.

Contributions. In this paper, we compare the PD^2 Pfair algorithm to the EDF-FF partitioning scheme, which uses "first fit" (FF) as a partitioning heuristic and EDF for per-processor scheduling. We begin by showing how to account for various system overheads in the schedulability tests for both approaches. We then present experimental results that show that PD^2 is a viable alternative to EDF-FF. We end by discussing several additional benefits of Pfair scheduling, such as simple and efficient synchronization, temporal isolation, fault tolerance, and support for dynamic tasks. Achieving these benefits under EDF-FF requires additional mechanisms, which can only increase overhead.

The remainder of this paper is organized as follows. Sec. 2 describes Pfair scheduling in detail. In Sec. 3, the state of partitioning research is summarized. Our comparison of PD^2 and EDF-FF is

a minimum separation. The notion of a rate-based task is a further generalization in which sporadic execution is allowed *within* a job. This is explained in more detail in Sec. 2 in the discussion of *intra-sporadic* tasks.

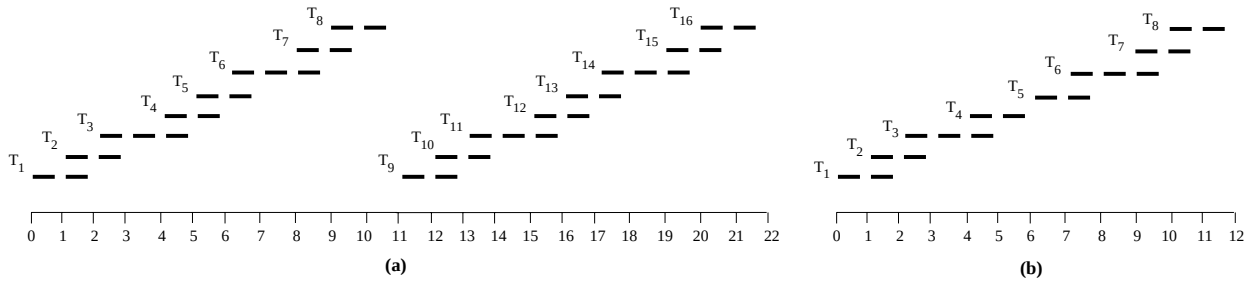


Figure 1: **(a)** Windows of the first two jobs of a periodic task T with weight 8/11. These two jobs consist of the subtasks $T_1, \ldots, T_8$ and $T_9, \ldots, T_{16}$, respectively. Each subtask must be scheduled during its window, or a lag-bound violation will result. **(b)** Windows of an IS task. Subtask T_5 becomes eligible one slot late.

presented in Sec. 4. In Sec. 5, we consider the additional advantages provided by Pfair scheduling noted above. Sec. 6 concludes the paper.

2 Pfair Scheduling

In defining notions relevant to Pfair scheduling, we limit attention (for now) to synchronous, periodic task systems. A periodic task T with an integer *period* $T.p$ and an integer *execution cost* $T.e$ has a *weight* $wt(T) = T.e/T.p$, where $0 < wt(T) \leq 1$. A task T is *light* if $wt(T) < 1/2$, and *heavy* otherwise.

Under Pfair scheduling, processor time is allocated in discrete time units, called *quanta*; the time interval $[t, t+1)$, where t is a nonnegative integer, is called *slot* t. (Hence, time t refers to the beginning of slot t.) In each slot, each processor can be allocated to at most one task. A task may be allocated time on different processors, but not within the same slot (*i.e.*, migration is allowed but parallelism is not). The sequence of allocation decisions over time defines a *schedule* S. Formally, $S\colon \tau \times \mathcal{N} \mapsto \{0, 1\}$, where τ is the set of N tasks to be scheduled and $\mathcal{N}$ is the set of nonnegative integers. $S(T, t) = 1$ iff T is scheduled in slot t. In any M-processor schedule, $\sum_{T \in \tau} S(T, t) \leq M$ holds for all t.

Lags and subtasks. A Pfair schedule is defined by comparing to an ideal fluid schedule in which $wt(T)$ processor time is allocated to each task T in each slot. Deviation from this fluid schedule is formally captured by the concept of *lag*. Formally, the *lag of task* T *at time* t is given by $lag(T, t) = wt(T) \cdot t - \sum_{u=0}^{t-1} S(T, u)$. A schedule is *Pfair* iff

$$(\forall T, t :: -1 < lag(T, t) < 1). \tag{1}$$

Informally, the allocation error associated with each task must always be less than one quantum.

Due to (1), each task T is effectively divided into an infinite sequence of quantum-length *subtasks*. The i^{th} subtask ($i \geq 1$) of task T is denoted T_i. As in [5], each subtask T_i has a *pseudo-release* $r(T_i)$ and *pseudo-deadline* $d(T_i)$, as specified below. (For brevity, we often omit the prefix "pseudo-".)

$$r(T_i) = \left\lfloor \frac{i-1}{wt(T)} \right\rfloor \qquad d(T_i) = \left\lceil \frac{i}{wt(T)} \right\rceil$$

T_i must be scheduled in the interval $w(T_i) = [r(T_i), d(T_i))$, termed its *window*, or (1) will be violated. Note that $r(T_{i+1})$ is either $d(T_i) - 1$ or $d(T_i)$. Thus, consecutive windows of the same task either overlap by one slot or are disjoint. The *length* of T_i's window, denoted $|w(T_i)|$, is $d(T_i) - r(T_i)$. As an example, consider subtask T_1 in Fig. 1(a). Here, we have $r(T_1) = 0$, $d(T_1) = 2$, and $|w(T_1)| = 2$. Therefore, T_1 must be scheduled in either slot 0 or slot 1.

Pfair scheduling algorithms. At present, three Pfair scheduling algorithms are known to be optimal on an arbitrary number of processors: PF [5], PD [6], and PD^2 [2]. These algorithms prioritize subtasks on an earliest-pseudo-deadline-first basis, but differ in the choice of tie-breaking rules. (Selecting appropriate tie-breaks turns out to be *the* most important concern in designing correct Pfair algorithms.) PD^2, which is the most efficient of the three, uses two tie-break parameters.

The first PD^2 tie-break is a bit, denoted $b(T_i)$. As seen in Fig. 1(a), consecutive windows of a Pfair task are either disjoint or overlap by one slot. $b(T_i)$ is defined to be 1 if T_i's window overlaps T_{i+1}'s, and 0 otherwise. For example, in Fig. 1(a), $b(T_i) = 1$ for $1 \leq i \leq 7$ and $b(T_8) = 0$. PD^2 favors subtasks with b-bits of 1. Informally, it is better to execute T_i "early" if its window overlaps that of T_{i+1}, because this potentially leaves more slots available to T_{i+1}.

The second PD^2 tie-break, the *group deadline*, is needed in systems containing tasks with windows of length two. A task T has such windows iff $1/2 \leq wt(T) < 1$. Consider a sequence $T_i, \ldots, T_j$ of subtasks of such a task T such that $b(T_k) = 1 \ \wedge \ |w(T_{k+1})| = 2$ for all $i \leq k < j$. Scheduling T_i in its last slot forces the other subtasks in this sequence to be scheduled in their last slots. For example, in Fig. 1(a), scheduling T_3 in slot 4 forces T_4 and T_5 to be scheduled in slots 5 and 6, respectively. The group deadline of a subtask T_i is the earliest time by which such a "cascade" must end. Formally, it is the earliest time t, where $t \geq d(T_i)$, such that either $(t = d(T_k) \ \wedge \ b(T_k) = 0)$ or $(t + 1 = d(T_k) \ \wedge \ |w(T_k)| = 3)$ for some subtask T_k. For example, subtask T_3 in Fig. 1(a) has a group deadline at time 8 and subtask T_7 has a group deadline at time 11. PD^2 favors subtasks with later group deadlines because *not* scheduling them can lead to longer cascades.

Rate-based Pfair. Under Pfair scheduling, if some subtask of a task T executes "early" within its window, then T will be ineligible for execution until the beginning of its next window. This means that Pfair scheduling algorithms are necessarily not work conserving when used to schedule periodic tasks. (A scheduling algorithm is *work conserving* if processors never idle unnecessarily.) Work-conserving algorithms are of interest because they tend to improve job response times, especially in lightly-loaded systems. In addition, there is no need to keep track of when a job is and is not eligible.

In recent work, Anderson and Srinivasan proposed a work-conserving variant of Pfair scheduling called *"early-release" fair* (ERfair) scheduling [2, 4]. Under ERfair scheduling, if two subtasks are part of the same job, then the second subtask becomes eligible for execution as soon as the first completes. In other words, a subtask may be released "early," *i.e.*, before the beginning of its Pfair window. For example, if T_3 in Fig. 1(a) were scheduled in slot 2, then T_4 could be scheduled as early as time 3.

In other recent work, Anderson and Srinivasan extended the early-release task model to also allow subtasks to be released "late," *i.e.*, there may be separation between consecutive subtasks of the same task [3]. The resulting model, called the *intra-sporadic* (IS) model, generalizes the well-known sporadic model, which allows separation between consecutive jobs of the same task. An example of an intra-sporadic task is shown in Fig. 1(b). Note that T_5 is released one slot late. In [3], Anderson and Srinivasan proved that an IS task system τ is feasible on M processors iff

$$\sum_{T \in \tau} wt(T) \leq M. \tag{2}$$

In addition, they proved that PD^2 optimally schedules intra-sporadic task systems [39].

The IS model is suitable for modeling many applications, especially those involving packets arriving over a network. Due to network congestion and other factors, packets may arrive late or in bursts. The IS model treats these possibilities as first-class concepts. In particular, a late packet arrival corresponds to an IS delay. On the other hand, if a packet arrives early (as part of a bursty sequence), then its eligibility time may be less than its Pfair release time. Note that its Pfair release time determines its deadline. Thus, an early packet arrival is handled by "postponing" the subtask's deadline to where it

would have been had the packet arrived on time.

Dynamic task systems. In recent work, Srinivasan and Anderson [38] derived conditions under which IS tasks may dynamically join and leave a running system, without causing any missed deadlines. As expected, a task may join as long as (2) continues to hold. Leaving, however, is more complicated. (For example, if a task with negative lag is allowed to leave, and if it re-joins immediately, then it can effectly execute at a rate higher than its prescribed rate. Deadlines may be missed as a result.) If task T is light, then it can leave at or after time $d(T_i) + b(T_i)$, where T_i denotes its last-scheduled subtask. If T is heavy, then it can leave after its next group deadline.

3 Partitioning Approaches

Under partitioning, each task is assigned to a processor, on which it will exclusively execute. Finding an optimal assignment of tasks to processors is equivalent to a bin-packing problem, which is known to be NP-hard in the strong sense. Several polynomial-time heuristics have been proposed for solving this problem. Several are described below.

First Fit (FF): Each task is assigned to the first processor that can accept it. The uniprocessor schedulability test associated with the scheduling algorithm being used can be used as an acceptance test. For instance, under EDF scheduling, a task can be accepted on a processor as long as the total utilization of all tasks bound to that processor does not exceed unity.

Best Fit (BF): Each task is assigned to a processor that **(i)** can accept the task, and **(ii)** will have minimal remaining spare capacity after its addition.

First Fit Decreasing (FFD): FFD is same as FF, but tasks are considered in order of decreasing utilizations. ("Best Fit Decreasing" can be analogously defined.)

Although FF and BF can easily be used online, the use of more complex heuristics, such as FFD, may introduce unacceptable runtime overhead. For instance, under FFD, the set of all previously-accepted tasks must be re-sorted and re-partitioned each time a new task arrives. For similar reasons, many other available partitioning heuristics will likely be impractical for online scheduling.

Despite this, partitioning is beneficial in that, once tasks are assigned to processors, each processor can be scheduled independently using uniprocessor scheduling algorithms such as RM and EDF [8, 13, 27, 31, 36]. One of the most popular approaches is RM-FF, in which the FF heuristic is used to assign tasks to processors, which are scheduled using the RM algorithm. The popularity of this approach stems largely from the popularity of RM as a uniprocessor scheduling algorithm. One reason for RM's popularity is that it gives predictable performance for critical tasks, even under overload conditions.

One major problem with RM-FF, however, is that the total utilization that can be guaranteed on multiprocessors for *independent* tasks is only 41% [30]. (If resource sharing is required, which is very likely, then this bound will decrease further.) In addition, if the exact feasibility test (from [25]) is used instead of the well-known uniprocessor utilization bound of 69%, then the partitioning problem becomes a more complex bin-packing problem involving variable-sized bins. Such problems are avoided by EDF, so we use EDF-FF as the basis for our comparisons.

Surprisingly, the worst-case achievable utilization on M processors for all of the above-mentioned heuristics is only $(M + 1)/2$, even when EDF is used. To see why, note that $M + 1$ tasks, each with utilization $(1 + \epsilon)/2$, cannot be partitioned on M processors, regardless of the partitioning heuristic. As ϵ tends to 0, the total utilization of such a task system tends to $(M + 1)/2$.

Better utilization bounds can be obtained if per-task utilizations are bounded. Suppose that u_m is the maximum utilization of any task in the system. Then, any task can be assigned to a processor

that has a spare capacity of at least u_m. This implies that if a set of tasks is not schedulable, then every processor must have spare capacity of less than u_m. Hence, the total utilization of such a task set is more than $M(1-u_m)+u_m$. Equivalently, any task set with utilization at most $M-(M-1)u_m$ is schedulable. Lopez *et al.* [27] have shown that this bound can be improved. Specifically, they proved that the worst-case achievable utilization on M processors is $(\beta M+1)/(\beta+1)$, where $\beta=\lfloor 1/u_m \rfloor$.

4 Preemption-related Overheads

The schedulability tests described in Secs. 2 and 3 were derived under the assumption that various overheads such as context-switching costs are zero. In practice, such overheads are usually not negligible. To account for them, task execution costs can be inflated. In this section, we show how to do so for both PD^2 and EDF-FF. The specific overheads we consider are described next.

Scheduling overhead accounts for the time spent moving a newly-arrived or preempted task to the ready queue and choosing the next task to be scheduled. Overheads associated with preemptions can be placed in two categories. *Context-switching overhead* accounts for the time the operating system spends on saving the context of a preempted task and loading the context of the task that preempts it. The *cache-related preemption delay* of a task refers to the time required to service cache misses that a task suffers when it resumes after a preemption. Note that scheduling and context-switching overheads are independent of the tasks involved in the preemption, whereas the cache-related preemption delay introduced by a preemption depends on **(i)** whether the preempted task resumes on a different processor, and **(ii)** which tasks execute in the meantime.

In a tightly-coupled, shared-memory multiprocessor, the cost of a migration is almost identical to that of a preemption. However, there might be some additional cache-related costs associated with a migration. If a task resumes execution on the same processor after a preemption (*i.e.*, without migrating), then some of the data that it accesses might still be in that processor's cache. This is highly unlikely if it resumes execution on a different processor. Nevertheless, because our analysis of cache-related preemption delays (described and justified later) assumes a cold cache after every preemption, there is no need to distinguish between preemptions and migrations.

All of the above overheads cause execution delays that must be considered when determining schedulability. In the rest of this section, we provide analytical bounds and empirical estimates of these various overheads for both EDF-FF and PD^2. For simplicity, we assume that all tasks are independent and either periodic or sporadic. Further, we assume that PD^2 is invoked at the beginning of every quantum. When invoked, it executes on a single processor and then conveys its scheduling decisions to the other processors. (Under PD^2, we also assume that all tasks are migratory; that is, the supertasking approach described later in Sec. 5.5 is not used.)

Context-switching overhead. Under EDF, the number of preemptions is at most the number of jobs. Consequently, the total number of context switches is at most twice the number of jobs. Thus, context-switching overhead can be accounted for by simply inflating the execution cost of each task by $2C$, where C is the cost of a single context switch. (This is a well-known accounting method.)

Under PD^2, a job may suffer a preemption at the end of every quantum in which it is scheduled. Hence, if a job spans E quanta, then the number of preemptions suffered by it is bounded by $E-1$. Thus, context-switching overhead can be accounted for by inflating the job's execution cost by $E \cdot C$. (The extra C term bounds the context-switching cost incurred by the first subtask of the job.)

A better bound on the number of preemptions can be obtained by observing that when a task is scheduled in two consecutive quanta, it can be allowed to continue executing on the same processor. For example, consider a task T with a period that spans 6 quanta and an execution time that spans 5 quanta. Then, in every period of T there is only one quantum in which it is not scheduled. This

implies that each job of T can suffer at most one preemption. In general, a job of a task with period of P quanta and an execution cost of E quanta can suffer at most $P - E$ preemptions. Combining this with our earlier analysis, the number of context switches is at most $1 + \mathbf{min}(E - 1, P - E)$.

Scheduling overhead. Another concern regarding PD^2 is the overhead incurred during each invocation of the scheduler. In every scheduling step, the M highest-priority tasks are selected (if that many tasks have eligible subtasks), and the values of the release, deadline, b-bit, and group deadline for each scheduled task are updated. Also, an event timer is set for the release of the task's next subtask. When a subtask is released, it is inserted into the ready queue. In the case of EDF-FF, the scheduler on each processor selects the highest-priority job from its local queue; if this job is not the currently-executing job, then the executing job is preempted. If the executing job has completed, then it is removed from the ready queue. Further, when a new job is released, it is inserted into the ready queue and the scheduler is invoked.

The partitioning approach has a significant advantage on multiprocessors, since scheduling overhead does not increase with the number of processors. This is because each processor has its own scheduler, and hence the scheduling decisions on the various processors are made independently and in parallel. On the other hand, under PD^2, the decisions are made sequentially by a single scheduler. Hence, as the number of processors increase, scheduling overhead also increases.

To measure the scheduling overhead incurred, we conducted a series of experiments involving randomly-generated task sets. Fig. 2(a) compares the average[4] execution time of one invocation of PD^2 with that of EDF on a *single* processor. We used binary heaps to implement the priority queues of both schedulers. The number of tasks, N, ranged over the set $\{15, 30, 50, 75, 100, 250, 500, 750, 1000\}$; for each N, 1000 task sets were generated randomly, each with total utilization at most one. Then, each generated task set was scheduled using both PD^2 and EDF until time 10^6 to determine the average execution cost per invocation for each scheduler.

As the graph shows, the scheduling overhead of each algorithm increases as the number of tasks increases. Though the increase for PD^2 is higher, the overhead is still less than $8\mu s$. When the number of tasks is at most 100, the overhead of PD^2 is less than $3\mu s$, which is comparable to that of EDF.

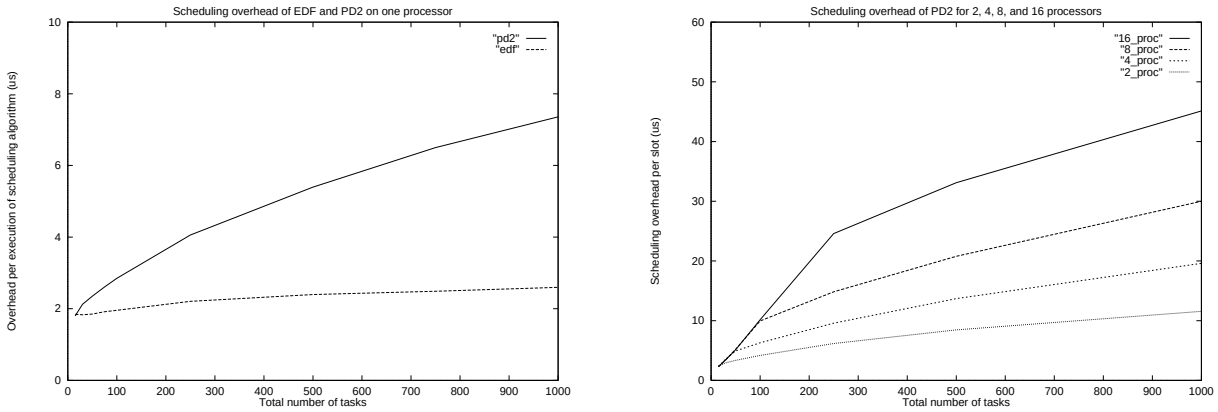
Fig. 2(b) shows the average scheduling overhead of PD^2 for $2, 4, 8$, and 16 processors. Again, the overhead increases as more tasks or processors are added. However, the scheduling cost for at most 200 tasks is still less than $20\mu s$, even for 16 processors. The cost of a context switch in modern processors is between 1 and 10 μs [34]. Thus, in most cases, scheduling costs are comparable to context-switching overheads under PD^2.

Scheduling overhead can be incorporated into a task's execution cost in much the same way as context-switching overhead. However, under PD^2, a job with an execution cost of E quanta incurs a scheduling cost at the beginning of every quantum in which it is scheduled. Hence, this cost is incurred exactly E times.

Cache-related preemption delay. For simplicity, we assume that *all* cached data accessed by a task is displaced upon a preemption or migration. Though this is clearly very conservative, it is difficult to obtain a more accurate estimate of cache-related preemption delay. This problem has been well-studied in work on timing analysis tools for RM-scheduled uniprocessors and several approaches have been proposed for obtaining better estimates [23, 24]. Unfortunately, no such techniques are available for either EDF or PD^2.

The cache-related preemption delay under EDF can be calculated as follows. Let $D(T)$ denote the maximum cache-related preemption delay suffered by task T. Let P_T denote the set of tasks that

[4]The experiments were performed on a 933-MHz Linux platform in which the minimum accuracy of the clock is $10ms$. Since the various scheduling overheads are much less than this value, averages are presented instead of maximum values.



(a) **(b)**

Figure 2: **(a)** Scheduling overhead of EDF and PD^2 on one processor. **(b)** Scheduling overhead of PD^2 on 2, 4, 8, and 16 processors. 99% confidence intervals were computed for each graph but are not shown because the relative error associated with each point is very small (less than 1.2% of the reported value).

are assigned to the same processor as T and that have periods larger than T's period. Note that T can only preempt tasks in P_T. To see why, consider a job J of T. Any job J' of T' that is executing prior to the release of J can be preempted by J only if J' has a later deadline. In that case, T' has a larger period than T's, *i.e.*, $T' \in P_T$. Also, job J can preempt at most one task in P_T: once J starts executing, no task in P_T will be scheduled by EDF until it completes. Thus, the overhead incurred due to the preemption of any task by J is bounded by $\mathbf{max}_{U \in P_T}\{D(U)\}$, and can be accounted for by inflating T's execution cost by this amount.

Under PD^2, the cache-related preemption delay of a task T is the product of $D(T)$ and the worst-case number of preemptions that a job of T can suffer (as derived earlier).

Simulation experiments. We now give formulae that show how to inflate task execution costs to include all of the above-mentioned overheads. Let S_A be the scheduling overhead per invocation of scheduling algorithm A. Let C denote the cost of a single context switch. Let P_T and $D(T)$ be as defined above. Let e and p be the execution time and period of task T. Let q denote the quantum size. (Then, the number of quanta spanned by t time units is $\lceil t/q \rceil$.) The inflated execution cost e' of task T can be computed as follows.[5] (We assume that p is a multiple of q.)

$$e' = \begin{cases} e + 2(S_{EDF} + C) + \mathbf{max}_{U \in P_T}\{D(U)\} & \text{, under EDF} \\ e + \left\lceil \frac{e'}{q} \right\rceil \cdot S_{PD^2} + C + \mathbf{min}(\left\lceil \frac{e'}{q} \right\rceil - 1, \frac{p}{q} - \left\lceil \frac{e'}{q} \right\rceil) \cdot (C + D(T)) & \text{, under PD}^2 \end{cases} \quad (3)$$

Using the above execution times, the schedulability tests are simple. For PD^2, we can use Equation (2). For EDF-FF, we invoke the first-fit heuristic to partition the tasks: since the new execution cost of a task depends on tasks on the same processor with larger periods, we consider tasks in the order of decreasing periods. (This dependency also complicates bin-packing in dynamic task systems.)

[5] In the formula for PD^2, e' occurs on the right-hand side as well because the number of preemptions suffered by a task may vary with its execution cost. e' can be easily computed by initially letting $e' = e$ and by repeatedly applying this formula until the value converges. Simulation experiments conducted by us on randomly-generated task sets suggest that convergence usually occurs within five iterations. However, more research is needed to obtain an analytic bound on the worst-case number of iterations.

To measure the schedulability loss caused by both system overheads and partitioning, we conducted a series of simulation experiments. In these experiments, the value of C is fixed at $5\mu s$. (As mentioned earlier, C is likely to be between 1 and $10\mu s$ in modern processors.) S_{EDF} and S_{PD^2} were chosen based on the values obtained by us in the scheduling-overhead experiments described earlier (refer to Fig. 2). $D(T)$ was chosen randomly between $0\mu s$ and $100\mu s$; the mean of this distribution was chosen to be $33.3\mu s$. We chose the mean of $33.3\mu s$ by extrapolating from results in [23, 24]. (Based on cache-cost figures reported by the authors of these papers, we estimated that the maximum cache-related preemption delay would be approximately $30\mu s$ in their experiments, assuming a bus speed of 66MHz.) The quantum size of the PD^2 scheduler was then assumed to be $1ms$.

The number of tasks, N, was chosen from the set $\{50, 100, 250, 500, 1000\}$. For each N, we generated random task sets with total utilizations ranging from $N/30$ to $N/3$, *i.e.*, the mean utilization of the tasks was varied from 1/30 to 1/3. In each step, we generated 1000 task sets with the selected total utilization. For each task set, we applied (3) to account for system overheads and then computed the minimum number of processors required by PD^2 and EDF-FF to render the task set schedulable.

Fig. 3(a) shows the averages of these numbers for $N = 50$. Note that when the total utilization is at most 3.75, both EDF and PD^2 give almost identical performance. EDF consistently gives better performance than PD^2 in the range $[4, 14)$, after which PD^2 gives slightly better performance. This is intuitive because when the utilization of each task is small, the overheads of PD^2 and EDF-FF are both negligible. As the utilizations increase, the influence of these overheads is magnified. Though the system overhead of EDF remains low throughout, the schedulability loss due to partitioning grows quickly, as can be seen in Fig. 4(a). After a certain point, this schedulability loss overtakes the other overheads. Note that, although EDF does perform better, PD^2 is always competitive.

The jagged nature of the lines in the graphs can be explained as follows. Recall that for each randomly-generated task set, we calculate the minimum number of processors required by each scheduling algorithm. For most of the task sets generated with a given total utilization, the number of processors required is identical. Hence, the average is close to an integer. As the total utilization increases, this average increases in spurts of one, resulting in jagged lines.

Insets (b)–(d) in Fig. 3 show similar graphs for $N = 100$, 250, and 500. Note that, for a given total utilization, mean utilization decreases as the number of tasks increases. As a result, the improvement in the performance of EDF-FF is more than that of PD^2. Therefore, the point at which PD^2 performs better than EDF-FF occurs at a higher total utilization.

Challenges in Pfair scheduling. One advantage of partitioning over Pfair scheduling is that researchers have studied it for more than 25 years, whereas Pfair scheduling is a relatively new topic. As a result, there are many open problems that need to be solved. We describe one of them below.

Though Pfair scheduling algorithms appear to be different from traditional real-time scheduling algorithms, they are similar to the round-robin algorithm used in general-purpose operating systems. In fact, PD^2 can be thought of as a deadline-based variant of the weighted round-robin algorithm. However, one difference is in the size of the scheduling quantum. Currently, for optimality, Pfair scheduling requires execution times to be a multiple of the quantum size. In systems in which this is not true, execution times must be rounded up to the next multiple of the quantum size. This can lead to a large loss in schedulability. For example, assuming a quantum size of 1, if a task has a small execution requirement of ϵ, then it must be increased to 1. (In the above experiments, this was one source of schedulability loss in PD^2.) If that task's period is 1, then this would mean a schedulability loss of $1 - \epsilon$ solely because of this task. This is clearly not acceptable in practice.

One way to reduce schedulability loss due to the above problem is to decrease the quantum size. There are two potential problems with this. First, the size of the quantum is constrained by the resolution of hardware clocks. Second, decreasing the quantum size increases schedulability loss due

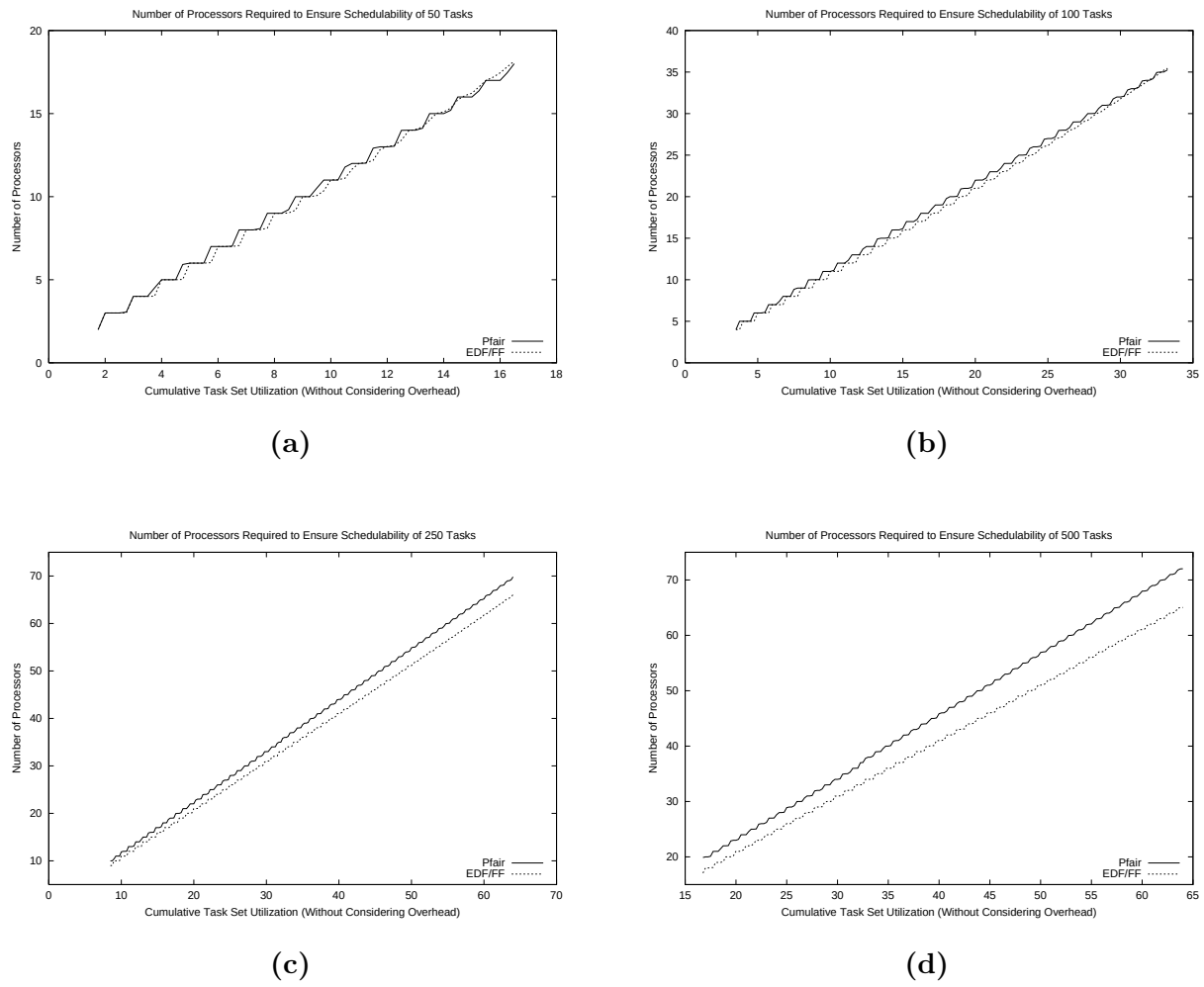


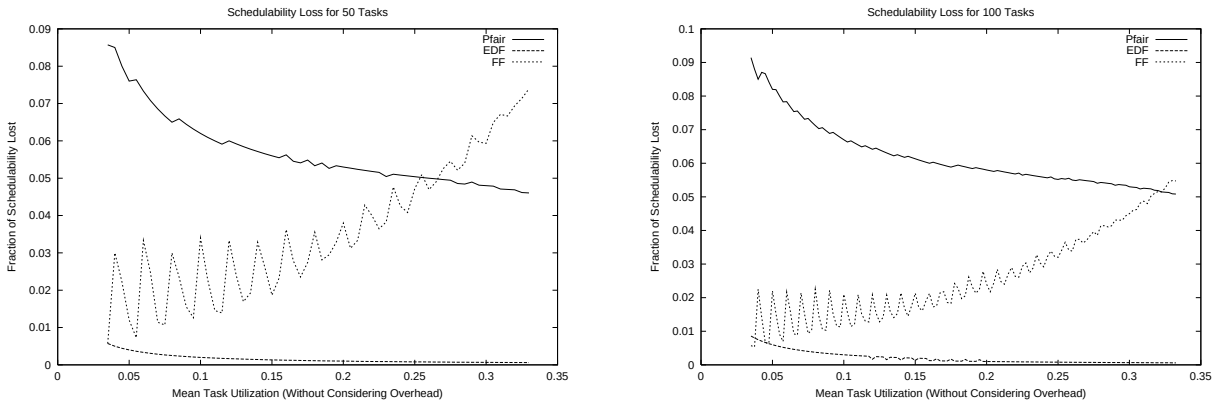
Figure 3: Effect of introducing overheads in the schedulability tests. Inset **(a)** shows the minimum number of processors that PD^2 and EDF require for a given total utilization of a system of 50 tasks. Insets **(b)**, **(c)**, and **(d)** show the same for 100, 250, and 500 tasks, respectively. As before, 99% confidence intervals were computed for each graph but are not shown because the relative error associated with each point is very small (less than 1% of the reported value).

to system overheads as evidenced by Equation (3). These trade-offs must be carefully analyzed to determine an optimal quantum size.

A more flexible approach is to allow a new quantum to begin immediately on a processor if a task completes execution on that processor before the next quantum boundary. However, with this change, quanta vary in length and may no longer align across all processors. It is easy to show that allowing such variable-length quanta can result in missed deadlines. Determining tight bounds on the extent to which deadlines might be missed remains an interesting open problem.

5 Benefits of Pfairness

In this section, we point out some of the inherent benefits of using Pfair scheduling algorithms.



(a) **(b)**

Figure 4: Fraction of schedulability loss due to partitioning and due to system overheads under PD^2 and EDF-FF for systems of **(a)** 50 and **(b)** 100 tasks. 99% confidence intervals were computed for each graph but, once again, are not shown. For each EDF and PD^2 point sample, relative error is very small (less than 4% and 2%, respectively). The relative error associated with the FF point samples is higher but drops off quickly as utilization increases; the error for these samples is initially approximately 17%, and is below 5% for all $U \geq 11$ in (a), and for all $U \geq 18$ in (b). The variation of adjacent FF sample values appears to be a fairly consistent indicator of the confidence interval's size at each point.

5.1 Resource and Object Sharing

In most systems, tasks need to communicate with external devices and share resources. Thus, it is not realistic to assume that all tasks are independent. To support non-independent tasks, schedulability tests are needed that take into account the use of shared resources. Such tests have been proposed for various synchronization schemes on uniprocessors, including the priority inheritance protocol [37], the priority ceiling protocol [37], the dynamic priority ceiling protocol [11], and EDF with dynamic deadline modification [19].

Under partitioning, if all tasks that access a common resource can be assigned to the same processor, then the uniprocessor schemes cited above can be used directly. However, resource sharing across processors is often inevitable. For example, if the total utilization of all tasks that access a single resource is more than one, then, clearly, it is impossible for all of them to be assigned to the same processor. Also, even if the total utilization of such tasks is at most one, one of the tasks may access other shared resources. It might not be possible to assign all tasks accessing those resources to the same processor.

Adding resource constraints to partitioning heuristics is a non-trivial problem. Further, such constraints also make the uniprocessor schedulability test less tight, and hence, partitioning less effective. The multiprocessor priority ceiling protocol (MPCP) has been proposed as a means to synchronize access to resources under partitioning [33]. Unfortunately, the MPCP was proposed only for RM-scheduled systems and needs to be adapted for use in EDF-scheduled systems. To the best of our knowledge, no multiprocessor synchronization protocols have been developed for partitioned systems with EDF (though it is probably not difficult to extend the MPCP to EDF-scheduled systems).

On the other hand, the tight synchrony in Pfair scheduling can be exploited to simplify task synchronization. Specifically, each subtask's execution is effectively non-preemptive within its time slot. As a result, problems stemming from the use of locks can be altogether avoided by ensuring that all locks are released before each quantum boundary. The latter is easily accomplished by delaying the start of critical sections that are not guaranteed to complete by the quantum boundary. When

critical-section durations are short compared to the quantum length, which is expected to be the common case,[6] this approach can be used to provide synchronization with very little overhead [17].

Pfair's tight synchrony also facilitates the use of *lock-free* shared objects. Operations on lock-free objects are usually implemented using "retry loops." Lock-free objects are of interest because they do not give rise to priority inversions and can be implemented with minimal operating system support. Implementation of such objects in multiprocessor systems has been viewed as being impractical because deducing bounds on retries due to interferences *across* processors is difficult. However, Holman and Anderson have shown that the tight synchrony in Pfair-scheduled systems can be exploited to obtain reasonably tight bounds on multiprocessors [18].

5.2 Dynamic Task Systems

Prior work in the real-time-sytems literature has focused mostly on static systems, in which the set of running tasks does not change with time. However, systems exist in which the set of tasks may change frequently. One example of such a system is a virtual-reality application in which the user moves within a virtual environment. As the user moves and the virtual scene changes, the time required to render the scene may vary substantially. If a single task is responsible for rendering, then its weight will change frequently. Task *reweighting* can be modeled as a leave-and-join problem, in which a task with the old weight leaves the system and a task with the new weight joins.

Implementing such systems by partitioning is problematic because the partitioning algorithm must be executed each time a new task joins. Hence, it can be costly to determine whether the new set of tasks is feasible. Of course, the efficient schedulability test for EDF-FF presented by Lopez *et al.* [27] (refer to Sec. 3) could be used, but its pessimism will likely require more processors than are actually necessary. On the other hand, adding tasks under PD^2 is simple (refer to Sec. 2).

5.3 Temporal Isolation

Fairness results in temporal isolation, *i.e.*, each task's processor share is guaranteed even if other tasks "misbehave" by attempting to execute for more than their prescribed shares. For this reason, fair (uniprocessor) scheduling mechanisms have been proposed in the networking literature as a means for supporting differentiated service classes in connection-oriented networks [7, 12, 14, 32, 40]. (Here, packet transmissions from various connections, or flows, are the "tasks" to be scheduled.) In addition, by using fair algorithms to schedule operating system activities, problems such as *receive livelock* [20] can be ameliorated.

Temporal isolation can be achieved among EDF-scheduled tasks by using additional mechanisms such as the constant-bandwidth server [1]. In this approach, the deadline of a job is postponed when it consumes its worst-case execution time. In other words, any overrun causes the excess execution to be "pushed" into the execution time reserved for later jobs. Though effective, the use of such mechanisms increases scheduling overhead.

5.4 Fault Tolerance and Overload

The presence of faults may cause some processors to become *overloaded*. In particular, if K out of M processors fail, then all tasks now need to be executed on the remaining $M - K$ processors. The

[6] In experiments conducted by Ramamurthy [35] on a 66 MHz processor, critical-section durations for a variety of common objects (*e.g.*, queues, linked lists, *etc.*) were found to be in the range of tens of microseconds. On modern processors, these operations will likely require no more than a few microseconds. Note that, in a Pfair-scheduled system, if such a critical section were preempted, then its length could easily be increased by several orders of magnitude.

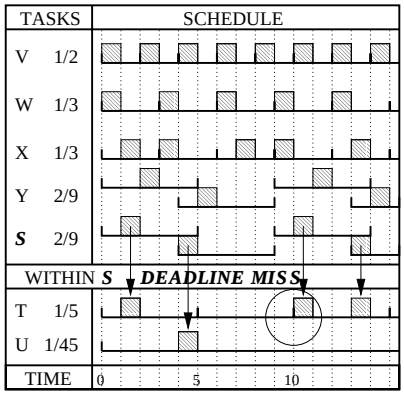


Figure 5: Sample PD^2 schedule for a task set consisting of six tasks with weights 1/2, 1/3, 1/3, 2/9, 1/5 and 1/45. The tasks of weight 1/5 and 1/45 are combined into a single supertask S, which competes with weight 2/9 and is bound to processor 2. The windows of each task are shown on alternating lines (*e.g.*, S's first window spans $[0, 5)$), and a shaded box denotes the quantum allocated to each subtask. The arrows show S passing its allocation (upper schedule) to one of its component tasks (lower schedule).

resulting system may be overloaded if total utilization is more than $M - K$. Overload can also occur under other unanticipated emergency conditions.

Pfair scheduling ensures graceful degradation of system performance in the presence of overload. If there are critical tasks in the system, then non-critical tasks can be reweighted to execute at a slower rate, thus ensuring that critical tasks are not affected by the overload. Further, in the special case in which total utilization is at most $M - K$, the optimality and global nature of Pfair scheduling ensures that the system can tolerate the loss of K processors transparently.

Under the partitioning approach, if a processor fails, then all tasks running on that processor need to re-assigned. Such a re-assignment may cause overloads on other processors, even when total utilization is less than $M - 1$. EDF has been shown to perform poorly under overload [15, 21]. Hence, additional mechanisms are required to maintain system stability under partitioning.

5.5 The Supertasking Approach

In [29], Moir and Ramamurthy observed that the migration assumptions underlying Pfair scheduling may be problematic. Specifically, tasks that communicate with external devices may need to execute on specific processors and hence cannot be migrated. They further noted that statically binding some tasks to specific processors may significantly reduce migration overhead in Pfair-scheduled systems.

To support non-migratory tasks, they proposed the use of *supertasks*. In this approach, each supertask replaces a set of *component tasks*, which are bound to a specific processor. Each supertask competes with a weight equal to the cumulative weight of its component tasks. Whenever a supertask is scheduled, one of its component tasks is selected to execute according to an internal scheduling algorithm. Unfortunately, as shown by Moir and Ramamurthy, component-task deadline misses may occur when using supertasks with PF, PD, or PD^2.

Fig. 5 depicts a two-processor PD^2 schedule in which supertasking fails. In this schedule, there are four normal tasks V, W, X, and Y with weights 1/2, 1/3, 1/3, and 2/9, respectively, and one supertask, S, which represents two component tasks T and U, with weights 1/5 and 1/45, respectively. S competes with a weight of $1/5 + 1/45 = 2/9$. As the figure shows, T misses a deadline at time 10. This is because no quantum is allocated to S in the interval $[5, 10)$.

Recently, Holman and Anderson [16] showed that deadlines can be guaranteed by inflating supertask weights. If EDF scheduling is used within a supertask, then a sufficient weight inflation is $1/p$, where p is the smallest period among the supertask's component tasks. The problem of whether such an inflation is necessary still remains open.

The supertasking approach is attractive primarily because it combines the benefits of both Pfair scheduling and partitioning. (In fact, both EDF-FF and ordinary Pfair scheduling can be seen as special cases of the supertasking approach.) Context switching can be reduced by packing tasks into supertasks that use EDF internally and that are assigned very heavy weights. Since such a supertask is seldom preempted, the number of preemptions will approach that of an EDF-scheduled uniprocessor system. Supertasking is also useful because it can be applied to reduce overhead introduced by both locking and non-blocking forms of synchronization [17, 18]. This is because less-costly uniprocessor synchronization schemes can be applied within a supertask.

6 Concluding Remarks

Though optimal Pfair scheduling algorithms exist, the frequency of preemptions and migrations in such algorithms has led to some questions regarding their practicality. In this paper, we have investigated how preemption and migration overheads affect schedulability under the PD^2 Pfair algorithm, using the EDF-FF partitioning scheme as a basis for comparison. Our results show that, in all circumstances, PD^2 performs competitively. Furthermore, PD^2 provides additional benefits, such as efficient synchronization, temporal isolation, fault tolerance, and support for dynamic tasks. Mechanisms for providing similar benefits under EDF-FF can be costly (and, in some cases, such mechanisms do not even exist). If such mechanisms had been incorporated into both approaches in our experiments, EDF-FF would likely have performed much more poorly than PD^2. In other words, considering only independent tasks puts EDF-FF in the best light. Though more research needs to be done to improve the performance of Pfair scheduling algorithms, research conducted so far suggests that Pfair scheduling is indeed a viable alternative to partitioning for many systems.

References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proc. of the 19th IEEE Real-Time Systems Symposium*, pages 4–13, Dec. 1998.

[2] J. Anderson and A. Srinivasan. Early-release fair scheduling. In *Proc. of the 12th Euromicro Conference on Real-Time Systems*, pages 35–43, June 2000.

[3] J. Anderson and A. Srinivasan. Pfair scheduling: Beyond periodic task systems. In *Proc. of the 7th International Conference on Real-Time Computing Systems and Applications*, pages 297–306, Dec. 2000.

[4] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, pages 76–85, June 2001.

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

[6] S. Baruah, J. Gehrke, and C.G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proc. of the 9th International Parallel Processing Symposium*, pages 280–288, April 1995.

[7] J. Bennett and H. Zhang. WF^2Q: Worst-case fair queueing. In *Proc. of IEEE INFOCOM'96*, pages 120–128, March 1996.

[8] A. Burchard, J. Liebeherr, Y. Oh, and S. H. Son. Assigning real-time tasks to homogeneous multiprocessor systems. *IEEE Transactions on Computers*, 44(12):1429–1442, 1995.

[9] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors. In *Proc. of the Fourth ACM Symposium on Operating System Design and Implementation*, pages 45–58, Oct. 2000.

[10] A. Chandra, M. Adler, and P. Shenoy. Deadline fair scheduling: Bridging the theory and practice of proportionate-fair scheduling in multiprocessor servers. In *Proc. of the 7th IEEE Real-Time Technology and Applications Symposium*, pages 3–14, May 2001.

[11] M. Chen and K. Lin. Dynamic priority ceiling: A concurrency control protocol for real time systems. *Real-Time Systems*, 2(1):325–346, 1990.

[12] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proc. of the ACM Symposium on Communications Architectures and Protocols*, pages 1–12, 1989.

[13] S. Dhall and C. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[14] S. Golestani. A self-clocked fair queueing scheme for broadband applications. In *Proc. of IEEE INFOCOM*, pages 636–646, April 1994.

[15] J. Haritsa, M. Livny, and M. Carey. Earliest deadline scheduling for real-time database systems. In *Proc. of the 12th IEEE Real-Time Systems Symposium*, pages 232–243, 1991.

[16] P. Holman and J. Anderson. Guaranteeing Pfair supertasks by reweighting. In *Proc. of the 22nd IEEE Real-time Systems Symposium*, pages 203–212, December 2001.

[17] P. Holman and J. Anderson. Locking in Pfair-scheduled multiprocessor systems. In *Proc. of the 23rd IEEE Real-time Systems Symposium*, Dec 2002. To appear.

[18] P. Holman and J. Anderson. Object sharing in Pfair-scheduled multiprocessor systems. In *Proc. of the 14th Euromicro Conference on Real-Time Systems*, pages 111–120, June 2002.

[19] K. Jeffay. Scheduling sporadic tasks with shared resources in hard real-time systems. In *Proceedings of the 13th IEEE Symposium on Real-Time Systems*, pages 89–98, Dec. 1992.

[20] K. Jeffay, F.D. Smith, A. Moorthy, and J. Anderson. Proportional share scheduling of operating system services for real-time applications. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 480–491, Dec. 1998.

[21] E. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *Proc. of the 6th IEEE Real-Time Systems Symposium*, pages 112–122, 1985.

[22] S. Keshav. Private communication, 2001.

[23] C.-G. Lee, J. Hahn, Y.-M. Seo, S.-L. Min, R.Ha, S. Hong, C.-Y. Park, M. Lee, and C.-S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

[24] Chang-Gun Lee, Kwangpo Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Bounding cache-related preemption delay for real-time systems. *IEEE Transactions on Software Engineering*, 27(9):805–826, 2001.

[25] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proc. of the 10th IEEE Real-Time Systems Symposium*, pages 166–171, Dec. 1989.

[26] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[27] J. Lopez, M. Garcia, J. Diaz, and D. Garcia. Worst-case utilization bound for edf scheduling on real-time multiprocessor systems. In *Proc. of the 12th Euromicro Conference on Real-Time Systems*, pages 25–33, June 2000.

[28] J. Mogul and A. Borg. The effect of context switches on cache performance. In *Proc. of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, 1991.

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

[30] D. Oh and T. Baker. Utilization bounds for n-processor rate monotone scheduling with static processor assignment. *Real-Time Systems*, 15(2):183–192, 1998.

[31] Yingfeng Oh and Sang H. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Real-Time Systems*, 9(3):207–239, 1995.

[32] A. Parekh and R. Gallagher. A generalized processor sharing approach to flow-control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

[33] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proc. of the 9th IEEE Real-Time Systems Symposium*, pages 259–269. IEEE, 1988.

[34] R. Rajkumar. Private communication, 2002.

[35] S. Ramamurthy. A lock-free approach to object sharing in real-time systems, 1997. PhD thesis, University of North Carolina at Chapel Hill.

[36] S. Saez, J. Vila, and A. Crespo. Using exact feasibility tests for allocating real-time tasks in multiprocessor systems. In *Proceedings of the Tenth Euromicro Workshop on Real-time Systems*, pages 53–60, June 1998.

[37] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time system synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[38] A. Srinivasan and J. Anderson. Fair scheduling of dynamic task systems on multiprocessors. Manuscript, Sept. 2002.

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

[40] L. Zhang. Virtual clock: A new traffic control algorithm for packet-switched networks. *ACM Transactions on Computer Systems*, 9(2):101–124, May 1991.