Adapting Pfair Scheduling for Symmetric Multiprocessors

Philip Holman and James H. Anderson

Abstract— **We consider the implementation of a Pfair real-time scheduler on a symmetric multiprocessor (SMP). Although SMPs are in many ways well-suited for Pfair scheduling, experimental results presented herein suggest that bus contention resulting from the simultaneous scheduling of all processors can substantially degrade performance. To correct this problem, we propose a *staggered* model for Pfair scheduling that strives to improve performance by more evenly distributing bus traffic over time. Additional simulations and experiments with a scheduler prototype are presented to demonstrate the effectiveness of the staggering approach. In addition, we discuss other techniques for improving performance *while maintaining worst-case predictability*. Finally, we present an efficient scheduling algorithm to support the proposed model and briefly explain how existing Pfair results apply to staggered scheduling.**

Keywords: SMP, Pfairness, real-time, scheduling, bus, multiprocessor

I. Introduction

In traditional systems, programs (or *tasks*) must exhibit *logical* correctness. Informally, logical correctness requires that each task produce appropriate results when executed, relative to its specification. Real-time systems strengthen this notion of correctness by also requiring *temporal* correctness. Informally, temporal correctness constrains each task invocation (or *job*) to execute in a *predictable* manner so that specified timing constraints are met. For instance, the most common form of timing constraint is a job *deadline*, which specifies the latest time by which that job must complete.

When sharing processors among a collection of real-time tasks, the choice of scheduling policy is key to the correct operation of the system. Systems can be characterized as either *soft* or *hard*, depending on whether some timing violations can be tolerated at runtime or not, respectively. In the case of hard real-time systems, analytical guarantees are needed to ensure that timing violations cannot occur, regardless of runtime conditions. Such guarantees are provided by showing that all constraints will be met even under *worst-case* conditions.

In this paper, we consider task scheduling in hard real-time multiprocessor systems. Multiprocessor scheduling techniques fall into two general categories: *partitioning* and *global scheduling*. In the partitioning approach, each processor schedules tasks independently from a local ready queue. When a new task arrives, it is assigned to one of these ready queues and executes only on the associated processor. In contrast, all ready tasks are stored in a single queue under global scheduling. Since 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. Whereas partitioning avoids moving tasks between processors (called *migration*), global scheduling may result in frequent migration due to the use of a shared queue.

At present, partitioning is the preferred approach on real-time multiprocessors, largely because it has been well-studied and has performed reasonably well in practice. Despite this, recent research has shown that global scheduling provides many advantages over partitioning approaches, including improved schedulability[1] and flexibility [1], [2], [19]. Furthermore, these benefits can be achieved without incurring significantly more (worst-case) overhead [21]. However, there remain questions as to whether global scheduling can achieve comparable average-case performance.

Though partitioning provides many performance advantages, such as improved cache performance (on average) and low scheduling overhead, it is inherently suboptimal: some systems are schedulable only when migration is permitted. In addition, optimal partitioning methods are costly, *i.e.*, partitioning is a variation of the bin-packing problem, which is known to be NP-hard in the strong sense [7]. Thus, in on-line settings, suboptimal heuristics must be employed. Furthermore, the actual benefit obtained from characteristics like improved cache performance can be difficult to determine analytically. Hence, many of the advantages of partitioning do not benefit worst-case analysis.

Proportionate-fair (Pfair) scheduling [4] is a particularly promising global-scheduling approach. Indeed, Pfair scheduling is presently the only known optimal means for scheduling periodic [15], sporadic [17], and rate-based tasks[2] on a real-time multiprocessor [19]. Hence, Pfair scheduling is seemingly well-suited for systems in which worst-case predictability is required. Unfortunately, some aspects of Pfair scheduling may degrade performance and have led to questions regarding its practicality. These aspects are discussed in detail below.

Characteristics. First, Pfair scheduling is based on synchronized *tick* (or *quantum-based*) scheduling. Scheduling points occur periodically and, at each point, all processors are

Work supported by NSF grants CCR 9988327, ITR 0082866, CCR 0204312, and CCR 0309825, and by a grant of software from the QNX corporation. Some content was previously presented in preliminary form at the 10th IEEE Real-time Technology and Applications Symposium [13].

[1]A task set is *schedulable* under an approach if that approach can guarantee that all timing constraints are met.

[2]A *periodic* (respectively, *sporadic*) task is repeatedly invoked to generate a sequence of identical jobs; invocations occur with a known fixed (respectively, minimum) separation. *Rate-based* tasks exhibit more variability in their instantaneous rate of execution, but still execute at a consistent rate over long intervals.

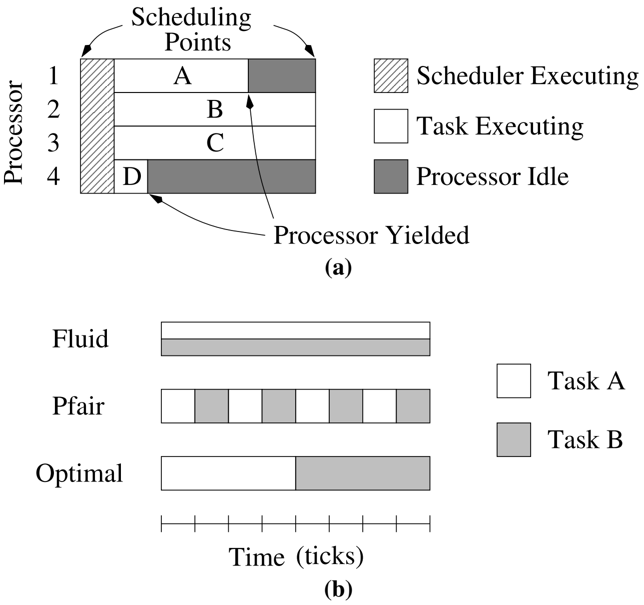


Fig. 1. **(a)** Two of the four scheduled tasks yield before the next scheduling point. **(b)** Relationship between a Pfair schedule (middle), the corresponding fluid schedule (upper), and a schedule that minimizes context switching (lower).

simultaneously scheduled. If a scheduled task yields before the next scheduling point, then that task is still charged for the unused processor time and the processor is idled. Figure 1(a) illustrates this characteristic.

Second, Pfair scheduling uses *weighted round-robin* scheduling to track the allocation of processor time in a *fluid* schedule, *i.e.*, a schedule in which each task executes at a constant rate. This achieves theoretical optimality, but at the cost of more frequent context switching. In practice, this cost is undesirable since it may increase scheduling overhead (depending on the quantum size) and reduces cache performance. Figure 1(b) shows a Pfair schedule, the corresponding fluid schedule, and a schedule with minimal switching.

Finally, task migration is unrestricted under Pfair scheduling. In the worst case, a task may be migrated each time it is scheduled. In practice, this may also negatively impact cache performance.

Prior work. In recent work, we have extended Pfair scheduling to address the above concerns and to enable its implementation and eventual comparison to partitioning. These extensions include task synchronization mechanisms [10], [11], techniques for accounting for system overheads [21], and support for hierarchal scheduling [9], [10], [11], [12]. To evaluate Pfair scheduling and its extensions, we have developed a Pfair prototype (from which we obtained some of the results presented later) that runs on a bus-based symmetric multiprocessor (SMP). This choice of platform follows from the fact that tight coupling is needed to keep preemption and migration overheads low. In addition, the worst-case cost of a migration is effectively the same as that of a preemption under a cache-based SMP. (Both worst-case scenarios correspond to

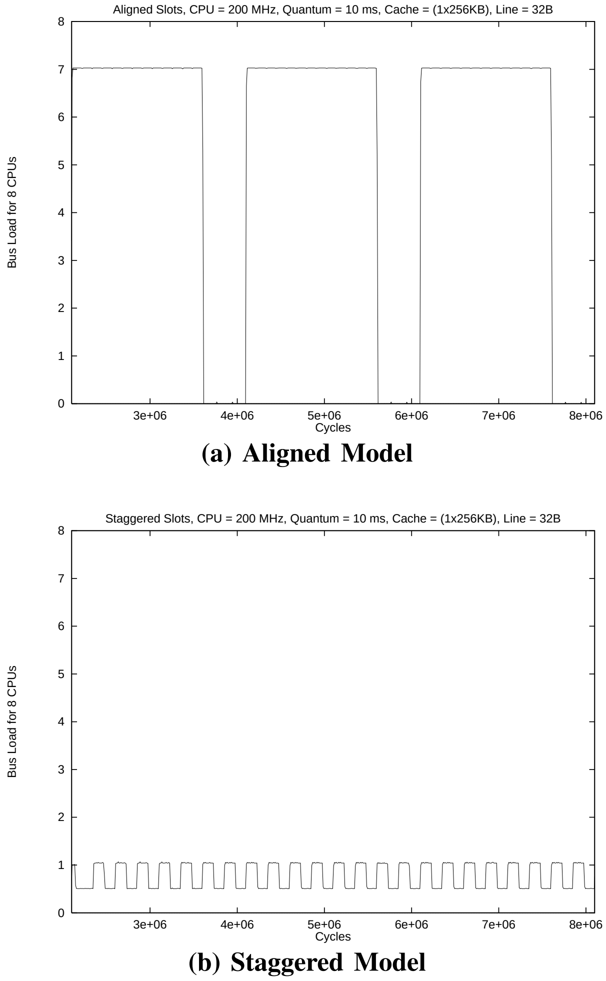


Fig. 2. Each graph shows the bus load across three slots in an eight-processor system using blocking caches. Graphs show contention **(a)** under the aligned model, the **(b)** under the (proposed) staggered model.

the case in which the local processor cache is cold, but filled with dirty lines.)

Contributions of this paper. In this paper, we show how Pfair scheduling actually *promotes* bus contention, and then propose an alternative scheduling model that strives to avoid this problem. The contention problem stems from the fact that a preempted task may encounter a cold cache when it resumes. Since bus traffic increases while reloading data into the cache, scheduling *all* processors simultaneously can result in very heavy bus contention at the start of each quantum. The worst-case duration of this contention grows with both the processor count and working-set sizes.

Figure 2(a) illustrates this contention on eight processors. The number of pending bus requests across three scheduling points (*i.e.*, spanning three quanta) are shown. In this experiment, each task was given an array that matched the cache's size and simply wrote to each cache line in sequence. (This experiment is considered in detail later.) As shown, heavy contention follows each scheduling point. Other results, presented later, suggest that such contention can significantly lengthen the execution times of tasks.

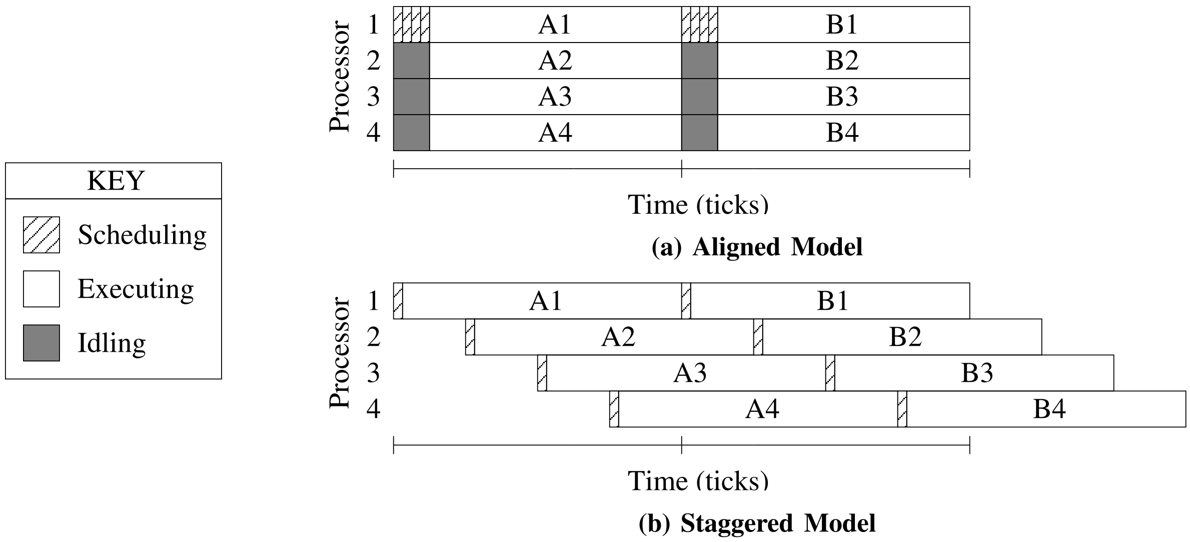


Fig. 3. **(a)** Under the aligned model, processors are simultaneously scheduled and all decisions are made on a single processor. **(b)** Under the staggered model, scheduling points are uniformly distributed and each processor can make its own scheduling decision.

The primary contribution of this paper is a new *staggered* model for Pfair scheduling that more uniformly distributes these predictable bursts of bus traffic. Results of simulations and experimental measurements obtained from our prototype system suggest that this model can significantly reduce bus contention and hence improve performance. This model provides the additional benefit that scheduling overhead can be distributed across processors, thereby reducing the per-processor overhead. Our second contribution is an efficient distributed scheduling algorithm for the staggered model and an evaluation of its performance. Finally, we characterize the impact of the new model on prior results and explain how these results can be modified for use under the new model.

Figure 3(b) shows the proposed staggered model along with the traditional *aligned* model. Repeating the earlier experiment using the staggered model results in dramatically lower contention, as shown in Figure 2(b). In addition, all scheduling decisions are made by processor 1 under the aligned model (see Figure 3(a)). As a result, cycles are lost due to stalling the other processors. Under the staggered model, each processor can make its own scheduling decision (see Figure 3(b)). By avoiding processor stalls, both the worst-case and average-case scheduling overhead is reduced.

Relevance. The problem of bus contention will obviously not be addressed solely by adopting this new model. If task behaviors are not restricted, then bus contention can also arise on-the-fly due to working-set changes that naturally occur within any task. Although we restrict attention in this paper to preemption-related bursts of bus traffic, we briefly sketch a more complete strategy for addressing the broader problem of bus contention below.

One approach for managing the bus that has already received much attention is time-division multiplexing. Under this approach, each processor is guaranteed exclusive access to the bus during statically chosen intervals of time. For instance, processor k could be guaranteed exclusive access to the bus at the start of each slot by reserving the interval $\left[i + \frac{k}{M}, i + \frac{k+1}{M}\right)$ for all $i \geq 0$. However, this also implies that processor k cannot access the bus for the remainder of each slot because the bus is reserved for other processors during that time. Under such a division, the interval $\left[i + \frac{k}{M}, i + \frac{k+1}{M}\right)$ acts as a *loading* phase for the i^{th} slot of processor k, during which the scheduled task should load the data it requires into the local processor cache. The remainder of the slot, *i.e.*, the interval $\left[i + \frac{k+1}{M}, i + 1 + \frac{k}{M}\right)$, then acts as a *work* phase, during which the task is permitted to manipulate only the cached data.

The primary disadvantage of this approach is that performance depends heavily on whether tasks can be restructured to respect such a restriction without wasting processor time. If a task is within its slot's work phase but requires uncached data to make further progress, then the task will be stalled until the next loading phase. As a result, the remainder of the current quantum will be wasted. Determining the degree of waste that is likely to occur is a daunting task since it requires fairly extensive knowledge of how real task sets are structured.

Another complication that impacts this approach is that performance will depend on the cache structure and coherence policy. Since bus traffic should not be generated during a work phase, a *write back* or similar policy is clearly necessary for this approach to be successful. Furthermore, low associativity is undesirable since it may prevent two segments of memory that map to the same cache line from co-existing in the cache, even if some lines in the cache are unused. Such restrictions reduce the amount of data that can be used during the work phase, and hence will likely increase waste.

In practice, it may be impractical to restrict bus access to a single processor, as assumed in the above discussion of time-division multiplexing. Due to operating system activities, such as the scheduling, it may be necessary to allow more than one processor to access the bus. However, such activities differ from task activities because the algorithms used by the operating system are known prior to runtime. Due to this increased knowledge, we expect the pessimism in the analysis to remain reasonable as long as access to the bus is restricted to at most one *task* at a given time. However, further study is certainly needed to determine if this expectation is accurate and to investigate the related issues discussed above.

Related work. Unfortunately, little (if any) prior work has considered techniques for improving performance in multiprocessor systems that require worst-case predictability. Consequently, available techniques are typically heuristic in nature and lack supporting analysis. Affinity-based scheduling algorithms are one common example. Such algorithms dynamically manipulate the priorities of tasks to make preemption less likely. By resisting preemption, these algorithms tend to improve cache performance, and hence reduce task execution times. Although the use of such techniques does improve average-case performance, the complexity of the priority definitions used makes worst-case analysis difficult. As a result, these techniques are not appropriate for hard real-time systems.[3] Though we are interested in average-case performance and soft real-time systems, our primary focus is on determining the inherent cost of guaranteeing predictable operation in multiprocessor systems. Techniques for optimizing Pfair scheduling for a general-purpose environment in which worst-case predictability is not required were previously proposed by Chandra, Adler, and Shenoy [6].

The remainder of the paper is organized as follows. Section II summarizes Pfair scheduling. An efficient scheduling algorithm for the staggered model is presented in Section III. In Section IV, we explain how to adapt prior results for use under the proposed model. Experimental results are then presented in Section V. We conclude in Section VI.

II. Background

In this section, Pfair scheduling is formally defined and previous work is summarized.

A. Basics of Pfair Scheduling

Let τ denote a set of N tasks to be scheduled on M processors. Each task $T \in \tau$ is assigned a rational *weight* $T.w$ in the range $(0, 1]$. Conceptually, $T.w$ is the rate at which T should be executed, relative to the speed of a single processor. (When scheduling periodic or sporadic tasks, each task is assigned a weight that approximately equals its utilization. See [8] for a detailed discussion of weight assignment.)

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 can be assigned to at most one task and each task can be assigned to at most one processor. Task migration is allowed. For simplicity, we assume that the quantum size is given. (We leave as future work the problem of selecting an optimal quantum size for a given system.)

Scheduling. Scheduling decisions are based on comparing each task's allocation to that granted in an ideal (fluid) system. Ideally, a task T receives $T.w \cdot L$ units of processor time in any interval of length L. This quantity is referred to as the *ideal* or *fluid* allocation. We let $\mathit{fluid}(T, t_1, t_2)$ denote the fluid allocation of T over the interval $[t_1, t_2)$. Formally, $\mathit{fluid}(T, t_1, t_2) = T.w \cdot (t_2 - t_1)$. (To facilitate the discussion, the equations presented herein do not account for scheduling overhead. Equations that do account for this overhead are presented in [8].)

The concept of tracking the ideal system is formalized by the notion of *lag*. Letting $\mathit{received}(T, t_1, t_2)$ denote the amount of processor time allocated to task T over the time interval $[t_1, t_2)$, the *lag* of T at time t can be formally defined as $\mathit{lag}(T, t) = \mathit{fluid}(T, 0, t) - \mathit{received}(T, 0, t) = T.w \cdot t - \mathit{received}(T, 0, t)$. A schedule respects Pfairness if and only if the magnitude of all task lags is strictly less than one always, *i.e.*, $(\forall T, t :: |\mathit{lag}(T, t)| < 1)$. As shown in [4], a schedule respecting Pfairness exists if and only if $\sum_{T \in \tau} T.w \leq M$.

Due to the use of a scheduling quantum, the above lag constraint effectively sub-divides each task T into a series of evenly distributed quantum-length *subtasks*. Let T_i denote the i^{th} subtask of T. Figure 4(a) shows the slot interval (or *window*) in which each subtask must execute to achieve Pfairness when $T.w = 3/10$. For example, T_2's window spans slots 3 through 6: T_2 is *released* at time 3 and has a *deadline* at time 7.

Schedulers. At present, PF [4], PD [5], and PD^2 [2] are the only known optimal Pfair scheduling algorithms. These algorithms prioritize subtasks on an earliest-deadline-first basis, but differ in the choice of (non-trivial) tie-breaking rules. Since the PD^2 prioritization is the most efficient, we assume its use. For our purposes, it is sufficient to know that PD^2 priorities can be determined and compared in constant time.

Model specifications. Let $t(i, k)$ denote the time (in quanta) at which the i^{th} scheduling point occurs on processor k, where $0 \leq k < M$. The aligned model is then defined by the expression $t(i, k) = i$, while the staggered model is defined by $t(i, k) = i + \frac{k}{M}$. We assume that the duration of processor scheduling under the staggered model does not exceed $\frac{1}{M}$.

B. Extensions

We now discuss extensions to Pfair scheduling. We consider adapting these extensions for use under the staggered model later in the paper.

Increased flexibility. Srinivasan and Anderson [1], [19] introduced three variants of Pfairness to improve scheduling flexibility. They also proved that PD^2 correctly schedules each variant whenever the cumulative task weight does not exceed M. *Early-release* fairness (ERfairness) allows subtasks to execute before their Pfair release times, provided that they are still prioritized by their Pfair deadlines. *Intra-sporadic* fairness (ISfairness) extends ERfairness by allowing windows to be right-shifted (*i.e.*, delayed relative to their Pfair placement). However, the relative separation between each pair of windows must be at least that guaranteed under Pfairness. Finally, *generalized intra-sporadic* fairness (GISfairness) extends ISfairness by allowing subtasks to be omitted. Figure 4 illustrates these

[3] Later, we consider a more limited use of processor affinity when assigning scheduled tasks to processors. This limited use differs from that described here in that task priorities are *not* dependent on processor affinity.

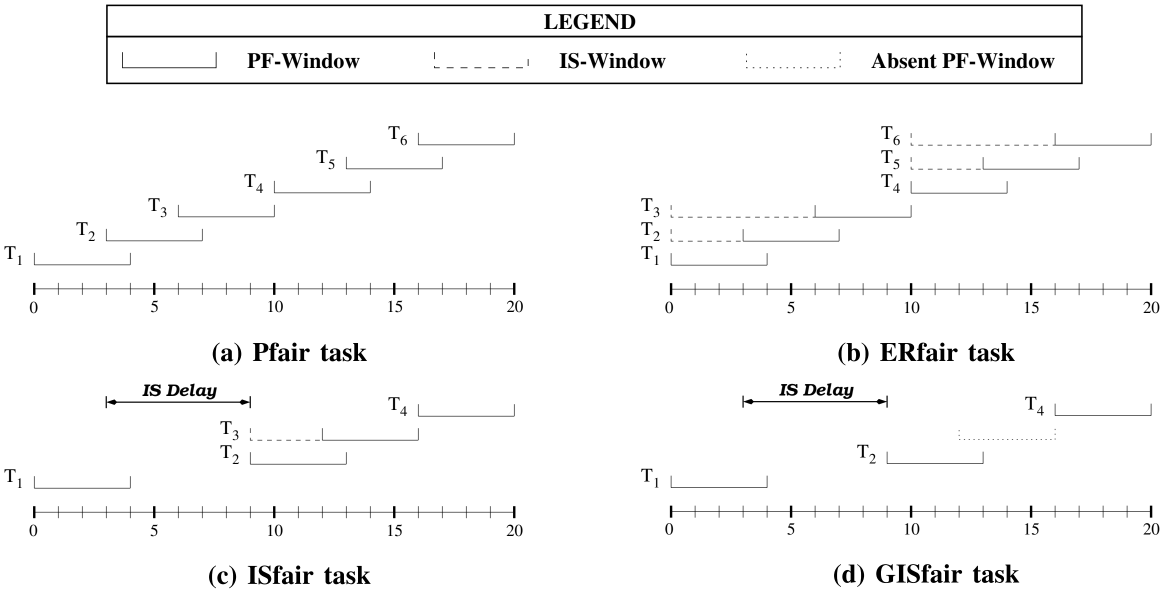


Fig. 4. The windowing for a task with weight $\frac{3}{10}$ is shown under the Pfair task model and its variants. **(a)** Normal windowing under the Pfair task model. **(b)** Early releasing has been used so that each grouping of three subtasks becomes eligible simultaneously. (In reality, each subtask will not be eligible until its predecessor is scheduled.) **(c)** Windows appear as in inset (b) except that T_2's release is now preceded by an intra-sporadic delay of six slots. (T_5 and T_6 are not shown.) **(d)** Windows appear as in inset (c) except that T_3's window is now absent.

variants. In addition to these variants, Srinivasan and Anderson also derived conditions under which tasks may leave and join a system [20].

Supertasking. *Supertasks* were first proposed to support *fixed* tasks, *i.e.*, tasks that must execute on a specific processor [16]. Each supertask represents a set of *component* tasks, which are scheduled as a single entity. Whenever a supertask is scheduled, one of its component tasks is selected to execute according to an internal scheduling algorithm. In previous work, we presented *reweighting* algorithms for deriving supertask weights from component task sets using either quantum-based [9] or fully preemptive [12] scheduling. (Our work does not restrict attention to the problem of scheduling fixed tasks.) In addition to supporting hierarchal scheduling, supertasks provide a means to selectively restrict which tasks may execute in parallel. Such restrictions can be imposed to reduce the worst-case contention for shared resources [10], [11]. Finally, the use of fully preemptive scheduling within supertasks can reduce loss caused by partially used quanta (see Figure 1(a)) by permitting other component tasks to consume unused time within the supertask's quanta.

Synchronization. In other prior work, we developed techniques for supporting lock-free and lock-based synchronization under Pfair scheduling [8], [10], [11]. For lock-based synchronization, we investigated two approaches: zone-based and server-based protocols. Zone-based protocols are designed to delay the start of short critical sections that are at risk of being preempted before completion. Hence, this approach exploits the quantum-based nature of Pfair scheduling to ensure that tasks will never hold locks while preempted. On the other hand, the server-based protocol uses a simple client-server model in which a server task executes all critical sections guarded by its associated locks.

C. Comparison with Partitioning

Conventional event-driven schedulers can usually be tested by replacing the schedulers in conventional operating systems, such as Linux or FreeBSD. However, due to its time-driven approach, forcing a Pfair scheduler into a conventional system will almost certainly produce both poor performance and measurements that are not reflective of a from-scratch implementation. Accurate assessment is essential to making an unbiased comparison to partitioning. Hence, mechanisms that exploit strengths and compensate for weaknesses are an important factor. Unfortunately, such mechanisms must first be developed for Pfair scheduling.

To determine whether such a (time-consuming) comparison was warranted, we conducted a study of scheduling overhead and schedulability loss based on analysis and simulation [21]. We found that the schedulability loss is comparable under both approaches and that Pfair scheduling does *not* incur prohibitively high scheduling overhead. We also noted several potential benefits to using Pfair scheduling, including efficient synchronization across processors, support for dynamic task sets, temporal isolation, and improved failure/overload tolerance. These results are promising since Pfair scheduling was proposed relatively recently and will likely improve significantly as it receives more attention. Partitioning, on the other hand, is well-studied and hence is not likely to improve.

III. Scheduling Algorithm

In this section, we present an efficient scheduling algorithm for the staggered model. For compactness, the presented algorithms use short-circuit condition evaluation, *i.e.*, we assume that the B term in $A \vee B$ is only evaluated when A is false.

A. Safety and Assignment Affinity

Unfortunately, staggered scheduling is slightly more complicated than traditional Pfair scheduling. Under staggering, quanta from different slots can overlap, as the A3 and B1 quanta illustrated in Figure 3(b). Hence, the scheduler must ensure that tasks scheduled *back-to-back* (*i.e.*, in consecutive slots) do not execute within overlapping quanta.

The algorithm that we present must ensure that this case is handled appropriately unless it can be shown that back-to-back scheduling *never* occurs. To disprove this latter claim, it is sufficient to consider the problem of avoiding back-to-back scheduling. Consider having to schedule a task x times. In order to avoid back-to-back scheduling, scheduling must be done over a range of at least $2x - 1$ slots. Since a task with weight $\frac{a}{b}$ must be scheduled a times over a range of b slots to satisfy Pfairness, it follows that any task with weight exceeding $\frac{x}{2x-1}$ for some integer $x > 0$ must be scheduled back-to-back. Furthermore, since $\lim_{x\to\infty} \frac{x}{2x-1} = \frac{1}{2}$, it follows that any task with weight exceeding $\frac{1}{2}$ must eventually be scheduled back-to-back. In addition, such heavyweight tasks are likely to occur in practice due, in part, to the use of techniques like supertasking. This observation suggests that back-to-back scheduling is likely to occur frequently in real systems, and hence that safety in the event of back-to-back scheduling is an essential requirement for correctness.

Affinity-based processor assignment. Safety can be ensured by employing an affinity-based policy when assigning scheduled tasks to processors. A variety of policies can be implemented by maintaining the small amount of scheduling history summarized below.

- **Previous Processor (PP)**: the most recent processor on which a task executed;
- **Previous Slot (PS)**: the most recent slot in which a task executed;
- **Previous Task (PT)**: the most recent task to execute on a processor.

Each of the above fields can be maintained by the scheduling algorithm at the cost of only a trivial increase in scheduling overhead. All of the following affinity-based policies, except for the last, can be implemented using one or more of the above fields.

- **Back To Back (BTB)**: When scheduled in consecutive slots, a task must be granted the same processor. Requires the PP and PS fields.
- **Last Scheduled (LS)**: Each processor must be assigned to the task to which it was last assigned, if that task is selected. Requires the PT field.
- **Most Recent Processor (MRP)**: Each selected task is assigned to the processor to which it was most recently assigned; if multiple tasks should be assigned the same processor, the tie is broken in favor of the task that most recently executed. Requires the PP and PS fields.
- **Most Recent Available Processor (MRAP)**: An extension of the MRP policy that considers processors beyond the most recently assigned processor. Specifically, when a task loses a tie for its desired processor, the task tries to execute on the processor on which it executed prior to the unavailable processor. This continues until the task is assigned a processor or its scheduling history is exhausted. This policy can be efficiently implemented by maintaining a list of processors in affinity order for each task, *i.e.*, the assigned processor is moved to the head of the list each time the task is scheduled. By using a doubly linked list, this list update can be implemented with constant time complexity.

Unfortunately, assignment policies are more difficult to implement under staggering, as we explain below. However, at least the BTB and LS policies can be implemented with minimal effort. Since the BTB policy is sufficient to prevent the allocation of overlapping quanta, we focus on it for the remainder of the paper.

In addition to ensuring safety under staggered scheduling, these policies are desirable because they can improve cache performance. When used with supertasks, the benefits of such a policy can be substantial, *i.e.*, the cache performance of a group of component tasks in a heavily weighted, fully preemptive supertask resembles that of using fully preemptive scheduling on a dedicated uniprocessor. Indeed, when using any of the policies described above, a unit-weight supertask effectively becomes a dedicated uniprocessor.

B. Concept

Before presenting the algorithm, we consider the problem of distributed scheduling with assignment affinity abstractly. By doing so, we intend to motivate the design of the algorithm presented later. Consider scheduling slot k on the first processor after executing task T on that processor in slot $k - 1$. To be computationally efficient, the scheduler must require only $O(\log N)$ time on each processor. (PD^2 schedules all M processors in $O(M \log N)$ time [19].) In addition, the decision should respect the BTB policy described above to ensure safety of the executing task, *i.e.*, if T is selected to execute in slot k, then it needs to execute on the first processor. However, simply identifying all tasks that are selected for slot k is an $\Omega(M)$ operation.

The implication is that the tasks scheduled in slot k must be identified *before* invoking the scheduler at the start of slot k. This can be achieved by dividing scheduling into two steps: **(i)** up to M tasks (if that many are eligible) are selected to execute in slot k and stored in k's *scheduling set*, and **(ii)** each processor (later) selects a task to execute in its local slot k from those in k's scheduling set. To ensure that k's scheduling set is known before slot k begins on any processor, Step (i) can be performed *one slot early* (*i.e.*, by the scheduler invocations associated with slot $k-1$). Specifically, processor p's scheduler invocation in slot $k - 1$ first selects a task to execute on processor p in slot $k-1$ from that slot's scheduling set, and then selects a task to add to the scheduling set of slot k (if an eligible task exists).

Necessity. Safety can actually be ensured by using a weaker policy than BTB. Specifically, the policy described below is both necessary and sufficient to ensure safety.

```
typedef task:
    record
        elig: integer;
        prio: ADT

shared var
    k: integer initially −1;
    SchedCount: integer initially M;
    Running: array 1 ... M of task ∪ {⊥}
                initially ⊥;
    SchedNow: min-heap of task initially ∅;
    ReschedNow: min-heap of task initially ∅;
    SchedNext: min-heap of task initially ∅;
    ReschedNext: min-heap of task initially ∅;
    Eligible: max-heap of task initially ∅;
    Incoming: array 0 ... ∞ of max-heap

private var
    p: 1 ... M; T: task ∪ {⊥}

procedure Initialize()
    Eligible := Eligible ∪ Incoming[0];
    while |Eligible| > 0 ∧ |SchedNext| < M do
        SchedNext :=
            SchedNext ∪ {ExtractMax(Eligible)}
    do

procedure SelectTask(T)
    if T ∈ S_k then
        ReschedNext := ReschedNext ∪ {T}
    else
        SchedNext := SchedNext ∪ {T}
    fi

procedure Schedule(p)
    if SchedCount = M then
        k := k + 1;
        Eligible := Eligible ∪ Incoming[k + 1];
        Swap(SchedNow, SchedNext);
        Swap(ReschedNow, ReschedNext)
    fi;
    if Running[p] ∈ ReschedNow then
        T := Running[p];
        ReschedNow := ReschedNow / {T}
    else if |SchedNow| > 0 then
        T := ExtractMin(SchedNow)
    else
        T := ⊥
    fi;
    Running[p] := T;
    if T ≠ ⊥ then
        UpdatePriority(T);
        if T.elig ≤ k + 1 then
            Eligible := Eligible ∪ {T}
        else
            Incoming[T.elig] :=
                    Incoming[T.elig] ∪ {T}
        fi
    fi;
    if |Eligible| > 0 then
        SelectTask(ExtractMax(Eligible))
    fi;
    SchedCount := (SchedCount mod M) + 1
```

Fig. 5. Basic staggered scheduling algorithm.

- **Directed Migration (DM)**: When scheduled back-to-back, a task that executed on processor p in the earlier slot must execute on a processor with index at least p in the later slot. Requires the PP and PS fields.

Under staggered scheduling, this policy provides an advantage over the BTB policy in that look-ahead scheduling, like that described above, is not necessary. However, because we are also interested in improving cache performance, we consider the BTB policy here.

Related work. As mentioned earlier, affinity-based policies are nothing new. Indeed, even assignment affinity has been considered in prior work. For instance, the (independently developed) preemption-aware dispatcher proposed by Andersson and Jonsson also focuses on providing a BTB guarantee [3]. Not surprisingly, the structure of their dispatcher strongly resembles the corresponding parts of our algorithm. However, in noting these similarities, it is important to keep in mind the goal of our algorithm: to distribute the scheduler overhead across processors. Though an interesting facet of the algorithm, assignment affinity is considered here only because it is the simplest method of ensuring safety under staggering.

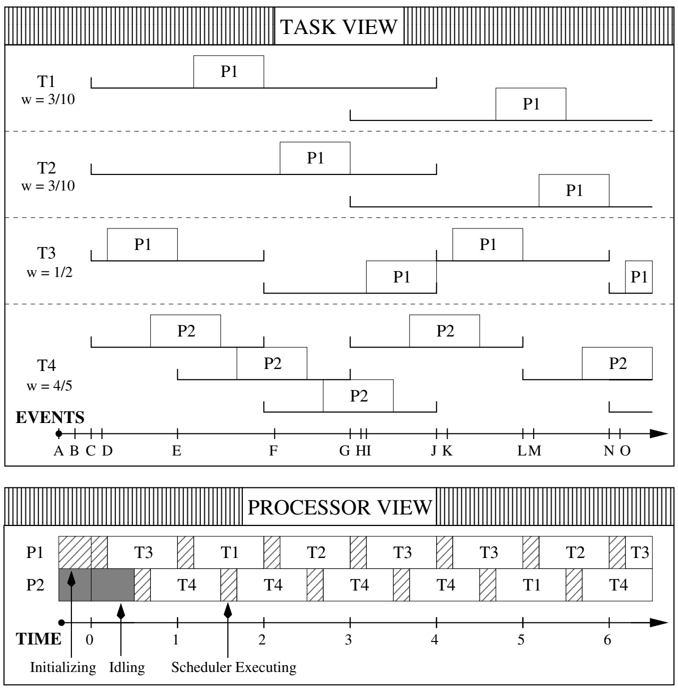
C. Basic Algorithm

We begin by presenting procedures (shown in Figure 5) needed to support Pfair or ERfair scheduling of static task sets. Later, we present additional procedures to support the remaining extensions to Pfair scheduling.

Data structures. Scheduling sets are implemented by the *SchedNow*, *ReschedNow*, *SchedNext*, and *ReschedNext* heaps. When scheduling slot k, tasks scheduled in slot k (respectively, $k+1$) are stored in the *Now* (respectively, *Next*) heaps. The *Sched* (respectively, *Resched*) heaps store tasks that are not (respectively, are) scheduled back-to-back. The *Eligible* heap stores all remaining tasks that are eligible in slot $k+1$. All heaps are ordered according to task priorities. (The $\preceq$ and $\succeq$ relations, which define this ordering, are defined below.) In addition, the *Incoming*[k] heap stores tasks that will not be eligible to execute until slot k. (We present these heaps as an unbounded array solely to simplify the presentation.) We assume that each task is initially stored in the appropriate *Incoming* heap.

Each task is represented by a record that contains (at least) the earliest slot in which the task may next execute (*elig*) and the task's current priority (*prio*). We assume that task priorities are implemented as an abstract data type that supports the $\prec$, $\preceq$, $\succ$, and $\succeq$ comparisons, where $\rho_1 \prec \rho_2$ (respectively, $\rho_1 \preceq \rho_2$) implies that priority ρ_1 is strictly higher than (respectively, at least as high as) priority ρ_2. We further assume that `UpdatePriority` encapsulates the algorithm for updating *prio* and *elig*.

The remaining variables include two counters (*k* and *Sched-*



T3's HEAP TRANSITIONS

EVENT	*DESTINATION*	*CAUSE OF TRANSITION*
A	Eligible	Initialize(): T3 is eligible to execute in slot 0.
B	SchedNext	Initialize(): T3 selected to execute in slot 0.
C	SchedNow	Schedule(P1): Slot 0 decisions committed.
D	Incoming[2]	Schedule(P1): T3 begins executing on P1 and its priority is updated (next eligible at time 2).
E	Eligible	Schedule(P1): T3 is eligible to execute in slot 2.
F	SchedNext	Schedule(P1): T3 is selected to execute in slot 3.
G	SchedNow	Schedule(P1): Slot 3 decisions committed.
H	Eligible	Schedule(P1): T3 begins executing on P1 and its priority is updated (next eligible at time 4).
I	ReschedNext	Schedule(P1): T3 is selected to execute in slot 4 and will execute back-to-back.
J	ReschedNow	Schedule(P1): Slot 4 decisions committed.
K	Incoming[6]	Schedule(P1): T3 begins executing on P1 and its priority is updated (next eligible at time 6).
L	Eligible	Schedule(P1): T3 is eligible to execute in slot 6.
M	SchedNext	Schedule(P1): T3 is selected to execute in slot 6.
N	SchedNow	Schedule(P1): Slot 6 decisions committed.
O	Incoming[8]	Schedule(P1): T3 begins executing on P1 and its priority is updated (next eligible at time 8).

Fig. 6. Four tasks (T1–T4) are executed on two processors (P1–P2) under the staggered model. **Left**: The schedule is shown from two different views. **Right**: Scheduling events affecting task **T3** are summarized in the table; event labels correspond to those shown on the timeline in the example schedule.

Count) and the *Running* array. k is the index of the current slot. *SchedCount* indicates the number of scheduler invocations that have been performed for slot *k*. Finally, the *Running* array indicates which task is currently executing on each processor.

To simplify the presentation, some branches test for set inclusion ($\in$) and the branch at line 4 uses S_k to denote the set of tasks selected to execute in slot k, *i.e.*, the scheduling set of k. All of these tests can be implemented in $O(1)$ time complexity and $O(N)$ space complexity by associating a few additional variables with each task.

Detailed description. `Initialize` is invoked before all other procedures to create slot 0's scheduling set. First, tasks present at time 0 are merged into the *Eligible* heap at line 1. Lines 2–3 then make the scheduling decisions.

`SelectTask` schedules a task T in slot $k+1$. Line 4 determines whether T is scheduled back-to-back. T is then stored in the proper heap in lines 5–6.

`Schedule`(*p*) is invoked to schedule processor *p*. Lines 8–11 initialize a round of scheduler invocations by incrementing *k*, by merging newly eligible tasks into *Eligible*, and by initializing *SchedNow*, *ReschedNow*, *SchedNext*, and *ReschedNext*. Lines 12–18 select the task to execute on processor p and record the decision. The task is then updated and stored according to the priority of its successor subtask in lines 20–23. Lines 24–25 then select a task to execute in the slot $k+1$. Finally, progress is recorded by updating *SchedCount* at line 26.

Example. Figure 6 shows a sample schedule produced by the staggered algorithm. To illustrate the operation of the algorithm, a trace of task T3's heap-membership changes is also shown.

D. Extensions

We now present the procedures needed to support tasks leaving and joining the system (shown in Figure 7). Systems consisting of dynamic task sets and those supporting ISfair and GISfair tasks require such support. In addition, systems that permit tasks to change weights at runtime will also require this support.[4]

Concerns. Two problems arise from tasks leaving and joining. First, these actions require modification of the scheduler's data structures. Hence, access to these structures must be synchronized.[5] Second, adding and removing tasks can lead to incorrect scheduling decisions. To avoid this potential problem, the presented procedures are designed to ensure that one of the two conditions given below holds after the execution of every procedure.

- **(I1)** $|\mathit{SchedNext}| + |\mathit{ReschedNext}| = \mathit{SchedCount}$ and, for all $T \in$ (*SchedNext* $\cup$ *ReschedNext*) and $U \in$ *Eligible*, $T.prio \preceq U.prio$.
- **(I2)** $|\mathit{SchedNext}| + |\mathit{ReschedNext}| < \mathit{SchedCount}$ and $|\mathit{Eligible}| = 0$.

Since $\mathit{SchedCount}$ records the number of completed scheduler invocations in the current round of decisions, it should be the case that $\mathit{SchedCount}$ tasks have been scheduled in the upcoming slot after each invocation. Informally, (I1) states that $\mathit{SchedCount}$ tasks have been tentatively scheduled in the

[4] A weight change can be achieved by having the task leave with the old weight and re-join with the new weight.

[5] The need for explicit synchronization can potentially be avoided by postponing handling of all leave and join requests until the next slot boundary. However, this approach has its own disadvantages in that postponed processing may lead to unnecessary idling and will also increase the overhead of boundary processing.

```
private var
    H: pointer to min-heap of task

procedure Deactivate(T)
27: if T ∈ SchedNext ∪ ReschedNext then
28:     if T ∈ ReschedNext then
29:         ReschedNext := ReschedNext / {T}
        else
30:         SchedNext := SchedNext / {T}
        fi;
31:     if |Eligible| > 0 then
32:         SelectTask(ExtractMax(Eligible))
        fi
33: else if T ∈ SchedNow ∪ ReschedNow then
34:     if T ∈ ReschedNow then
35:         ReschedNow := ReschedNow / {T}
        else
36:         SchedNow := SchedNow / {T}
        fi;
37:     UpdatePriority(T)
38: else if T ∈ Eligible then
39:     Eligible := Eligible / {T}
    else
40:     Incoming[T.elig] :=
            Incoming[T.elig] / {T}
    fi
```

```
procedure Activate(T)
41: if T.elig ≤ k + 1 then
42:     if |SchedNext| + |ReschedNext|
                < SchedCount then
43:         SelectTask(T)
        else
44:         if |SchedNext| = 0 ∨ (|ReschedNext| > 0
                ∧ Min(SchedNext).prio ⪯
                    Min(ReschedNext).prio) then
45:             H := &ReschedNext
            else
46:             H := &SchedNext
            fi;
47:         if T.prio ≺ Min(*H).prio then
48:             SelectTask(T);
49:             T := ExtractMin(*H)
            fi;
50:         Eligible := Eligible ∪ {T}
        fi
    else
51:     Incoming[T.elig] := Incoming[T.elig] ∪ {T}
    fi
```

Fig. 7. Extensions to support task departures (left) and task arrivals (right).

upcoming slot (first clause) and that each scheduled task has higher priority than each unscheduled task (second clause). In the event that too few eligible tasks exist, (I1) cannot be satisfied. This possibility is addressed by (I2). Informally, (I2) states that less than $SchedCount$ tasks have been tentatively scheduled (first clause), which suggests an idle processor in the upcoming slot, and that no other tasks are eligible for that slot (second clause). We explain how our algorithm guarantees that either (I1) or (I2) always holds by exhaustively considering all possible scenarios below.

Detailed description. Invoking `Deactivate`(T) in slot k causes task T to be ignored when scheduling slots at and after $k+1$. If T has been selected to execute in slot k but has not been granted a processor, then the decision to execute T is nullified; however, T is still charged as if it did execute. (This is analogous to having a task suspend immediately after it is granted the processor: the entire quantum is wasted.) When removing T, three cases must be considered to ensure that either (I1) or (I2) holds after execution: **(i)** T is scheduled in slot $k+1$ (it is in either *SchedNext* or *ReschedNext*), **(ii)** T is scheduled in slot k but has not been granted a processor (it is in either *SchedNow* or *ReschedNow*), and **(iii)** T is not scheduled in slot $k+1$ (but may be currently executing in slot k). Lines 28–32 handle Case (i) by locating and removing T from either *SchedNext* or *ReschedNext* at lines 28–30 and then scheduling a replacement task at lines 31–32. Lines 33–37 handle Case (ii) by locating and removing T from either *SchedNow* or *ReschedNow* at lines 33–36 and then charging T for the unused quantum at line 37. Lines 38–40 handle Case (iii).

Invoking `Activate`(T) within slot k causes task T to be considered when scheduling slots at and after $k+1$. Again, three cases must be considered to ensure that either (I1) or (I2) holds after execution: **(i)** T is not eligible to execute in slot $k+1$, **(ii)** T is eligible to execute in slot $k+1$ and a processor will idle in that slot, and **(iii)** T is eligible to execute in slot $k+1$ but no processor will idle in that slot. Lines 51 and 42–43 handle Cases (i) and (ii), respectively. Lines 44–50 handle Case (iii). Specifically, lines 44–46 determine which of *SchedNext* and *ReschedNext* contains the lowest-priority task that is scheduled in slot $k+1$. If this task's priority is lower than T's priority, then it is removed and replaced by T at lines 47–49. Whichever task is not scheduled in slot $k+1$ is then stored in *Eligible* at line 50.

Usage. Observe that `Deactivate` neither halts executing tasks nor modifies *Running*. The presented procedures are designed to be used as subroutines when implementing more complex services. Consequently, each procedure takes only those actions that are essential to achieving its goals. Indeed, it is not possible to present universal procedures for most services because their designs are based, at least in part, on policy decisions. For instance, a policy must be adopted to define how the system reacts to weight changes that cannot be immediately applied. Such policies are outside the scope of this discussion.

E. Time Complexity

Since task priorities can be updated and compared in constant time, the only significant complexity results from heap operations. Each procedure performs a constant number of heap operations and a constant number of calls to other procedures. Therefore, the time complexity of each procedure is $O(\log N)$ when using binomial heaps and PD^2 task priorities, and the aggregate time complexity of M scheduler invocations is $O(M \log N)$. Hence, the presented algorithm is computationally efficient.

Recall that under the aligned model, all scheduling decisions are made on a single processor, resulting in $O(M \log N)$ time complexity on that processor. The per-processor overhead under the staggered model is proportional to the time required by one invocation of `Schedule`, which has only $O(\log N)$ time complexity. This suggests that the staggered model can provide up to a factor-of-M improvement with respect to scheduling overhead. (The actual improvement will depend on the system architecture, as we later show.)

IV. Impact on Analysis

The most problematic aspect of shifting from an aligned to a staggered model is the impact on prior results. Though supporting the new model is reasonably straightforward (as we demonstrated in the last section), we must also consider the impact of staggering on analytical results and mechanisms proposed for Pfair scheduling. Unfortunately, it would be impractical to consider each prior result in detail here. For brevity, we discuss this issue more broadly in this section by focusing on the side effects produced by staggering. We then illustrate the adaptation of prior results by explaining the impact of these side effects on the extensions described earlier in Section II.

Interestingly, the shift from aligned to staggered quanta is a superficial change in that it does not actually impact the basic properties of Pfair algorithms. Under these algorithms, items are simply assigned to locations so that a given set of weight constraints are satisfied. (The problem is somewhat akin to a bin-packing problem using an infinite sequence of bins.) Weights serve only to restrict where the algorithm is allowed to assign each item. It is in the post-processing that we interpret items to be tasks and locations to correspond to fixed time intervals. The use of staggering impacts only the latter action. Consequently, staggering alters neither the operation of the algorithms nor their basic properties, such as the necessity of tie-breaking rules and the validity of known counterexamples. Hence, we need only be concerned in this section with the side effects produced by altering the interpretation of results produced by Pfair algorithms.

Side effects. By the specifications given earlier, a staggered slot extends up to $\Delta \overset{\text{def}}{=} \frac{M-1}{M}$ beyond the placement of the corresponding slot under the aligned model. Hence, slot k on a processor may overlap slots $k+1$ and $k-1$ on other processors, leading to the following side effects.

- **(E1)** An event occurring at time t under the aligned model may be delayed until time $t + \Delta$ under staggering.
- **(E2)** Each slot overlaps $M - 1$ other slots when aligned, but $2(M - 1)$ when staggered.

A. Independent Tasks

We begin by considering the scheduling of independent tasks, which do not synchronize or share resources. Independent tasks are oblivious to the concurrent execution of other tasks and, hence, are unaffected by (E2). (E1), on the other hand, has two implications, which are discussed below.

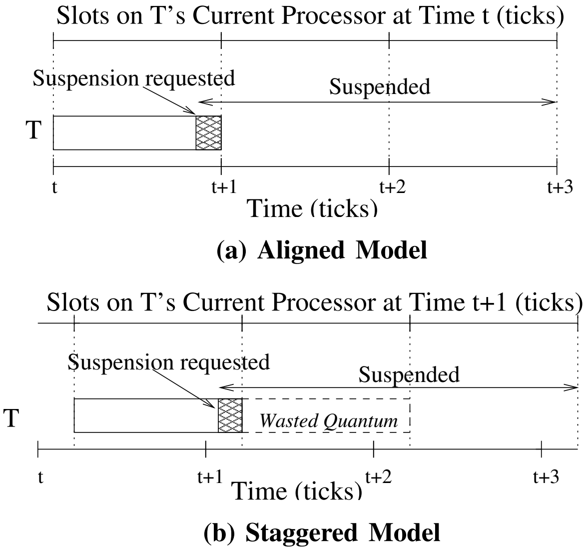


Fig. 8. Illustration of how event timing differs between **(a)** the aligned model and **(b)** the staggered model. An event in slot t is no longer guaranteed to occur before time $t + 1$ under the staggered model. Due to this, a task may have already been scheduled in the next slot before a suspension request is issued, as shown in (b).

First, deadlines guaranteed under the aligned model may be missed by up to Δ under staggering. We expect such misses, which are bounded by the slot length, to be acceptable in many cases. For instance, such a guarantee is inherent under proportional-share uniprocessor scheduling [22]. Strict deadlines can be enforced simply by treating task deadlines as if they are Δ units earlier than they actually are. The resulting loss depends on the task's parameters: a task T requiring a quanta every b slots will need $T.w \approx \frac{a}{b}$ under the aligned model, but $T.w \approx \frac{a}{b-1}$ under staggering. (This overhead can be more precisely characterized for specific tasks by adapting the analysis presented in [8].) This change in weight characterizes how staggering impacts the Pfair schedulability condition, which was presented earlier in Section II. As is common under Pfair scheduling, the penalty is not accessed to the right-hand side (M) of the condition, but rather to the left-hand side in the form of inflated task weights. In this case, the magnitude of the penalty is a function of the task weights prior to staggering. (We discuss how costly we expect this penalty to be at the end of the section.)

Second, events (such as suspension requests) occurring in slot k may occur *after* time $k+1$, at which point the scheduling decisions for slot $k+1$ are committed. As a result, a suspending task may occasionally cause a quantum to be wasted, as illustrated in Figure 8. (Recall that a quantum is wasted when a scheduled task leaves before being assigned a processor.)

Server tasks that suspend when no requests are pending will likely be most impacted by this property since worst-case analysis must pessimistically assume that a quantum is wasted each time the server suspends. Figure 9 depicts the worst-case scenario for a single request and response pair. Under the aligned model (see Figure 9(a)), task T sends a request to server S in slot 0. S then becomes active at time 1, services the request in slot 1, sends the response before time 2, and then suspends at time 2. Having received the response, T becomes ready at time 2 and is immediately scheduled in slot 2.

On the other hand, under the staggered model (see Figure 9(b)), T is scheduled on processor 2 in slot 0. As a result, its request is sent after time 1, at which point the scheduling decisions for slot 1 are committed. Because of this, T is scheduled in slot 1 even though it cannot make progress, and S remains suspended until time 2. T's suspension is finally processed at time 2. Upon becoming active, S is scheduled in slot 2 and executes on processor 3. S's response is then sent after time 3. As a result, T remains suspended until time 4, while S is re-scheduled in slot 3 and idles. T finally resumes at time 4.

In the staggering scenario, both the requesting task and the server waste a quantum (in slots 1 and 3, respectively). In addition, notice that the delay between the request and response is longer under staggering. Though unlikely, this worst-case scenario must be assumed to occur (under worst-case analysis) each time a request is issued unless other restrictions are placed on the runtime behavior of tasks.

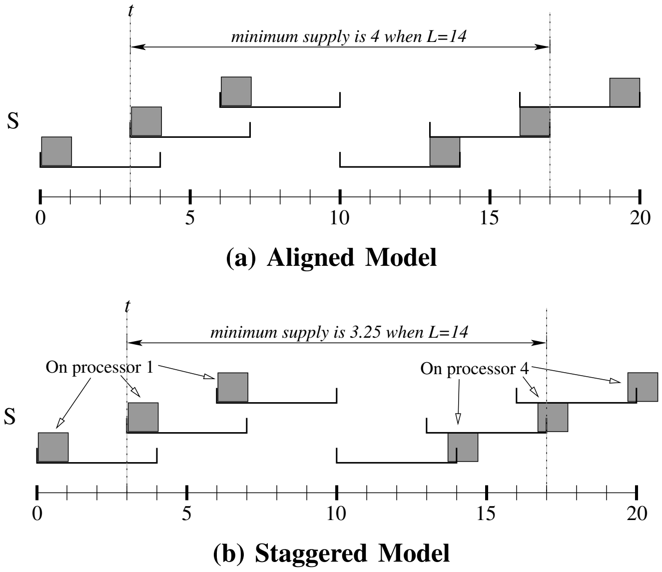


Fig. 10. The interval that defines the supply of a supertask (*i.e.*, minimum guaranteed allocation) when $M = 4$ (and hence $\Delta = \frac{3}{4} = 0.75$) and $L = 14$ is shown **(a)** under the aligned model and **(b)** under the staggered model. Due to staggering, the supply under the staggered model is Δ units lower than that under the aligned model.

B. Supertasking

Under the reweighting approach proposed by us in [8], [12], supertask weights are selected by comparing the least amount of processor time guaranteed to a supertask (called *supply*) to the maximum requirement of all component tasks (called *demand*). Only the first of these two quantities requires adjustment due to staggering. In addition, this quantity is unaffected by (E2).

The primary impact of (E1) is that the amount of processor time guaranteed to the supertask is slightly lower under staggering. In this case, the delay suggested by (E1) can be accounted for by reducing the estimate of each supertask's supply by Δ. Figure 10 shows how (E1) can impact the supply in this way. However, notice that this impact only occurs when the supertask is scheduled in the last slot of the interval, as depicted in Figure 10. Hence, staggering only affects estimates of supply that are based on such intervals. By taking this fact into account when deriving the supply function, tighter bounds can be obtained.

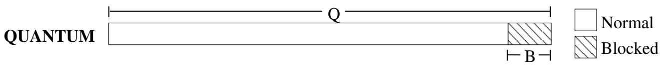
C. Lock-free Synchronization

Lock-free algorithms avoid locking by simply retrying operations until successful. Under lock-free analysis (*e.g.*, [10]), such operations are typically assumed to fail only if a concurrent operation succeeds. As a result, the worst-case number of retries is bounded by the worst-case number of concurrent operations.

Under the aligned model, it is sufficient to assume a worst-case mix of $M - 1$ interfering tasks when determining the worst-case number of retries. By (E2), similar reasoning can still be applied under staggering; however, $2(M - 1)$ tasks must be considered, which potentially doubles the overhead. It is possible to derive tighter bounds by considering that only a fraction of each of those $2(M-1)$ overlapping slots actually overlaps the slot in question. However, the resulting analysis would be much more complex (and time-consuming) than that presented in [10].

D. Zone-based Lock Synchronization

Zone-based protocols [8], [11] delay lock requests to prevent the preemption of short critical sections. This is accomplished by introducing an interval of *automatic blocking* at the end of each quantum, as illustrated below; when a critical section is at risk of being preempted before completion (due to starting late within a quantum), the associated lock request is ignored until a new quantum starts.



(In the diagram, Q denotes the quantum length, while B denotes the length of the automatic-blocking zone.)

Because the use of zones ensures that each critical section executes entirely in a single quantum, the analysis of these protocols is very similar to that of lock-free techniques [8], [10]: bounds on a task T's worst-case blocking overhead are computed by considering the interference of $M-1$ tasks under the aligned model and $2(M - 1)$ tasks under the staggered model. Unlike the lock-free case discussed above, there is little benefit provided by the property that only a fraction of each overlapping slot actually overlaps. This follows from the fact that each task that executes in parallel with the requesting task can interfere with (*i.e.*, block) the lock request only once under a zone-based approach. (Under lock-free analysis, a task can interfere with the same operation multiple times within the same slot.)

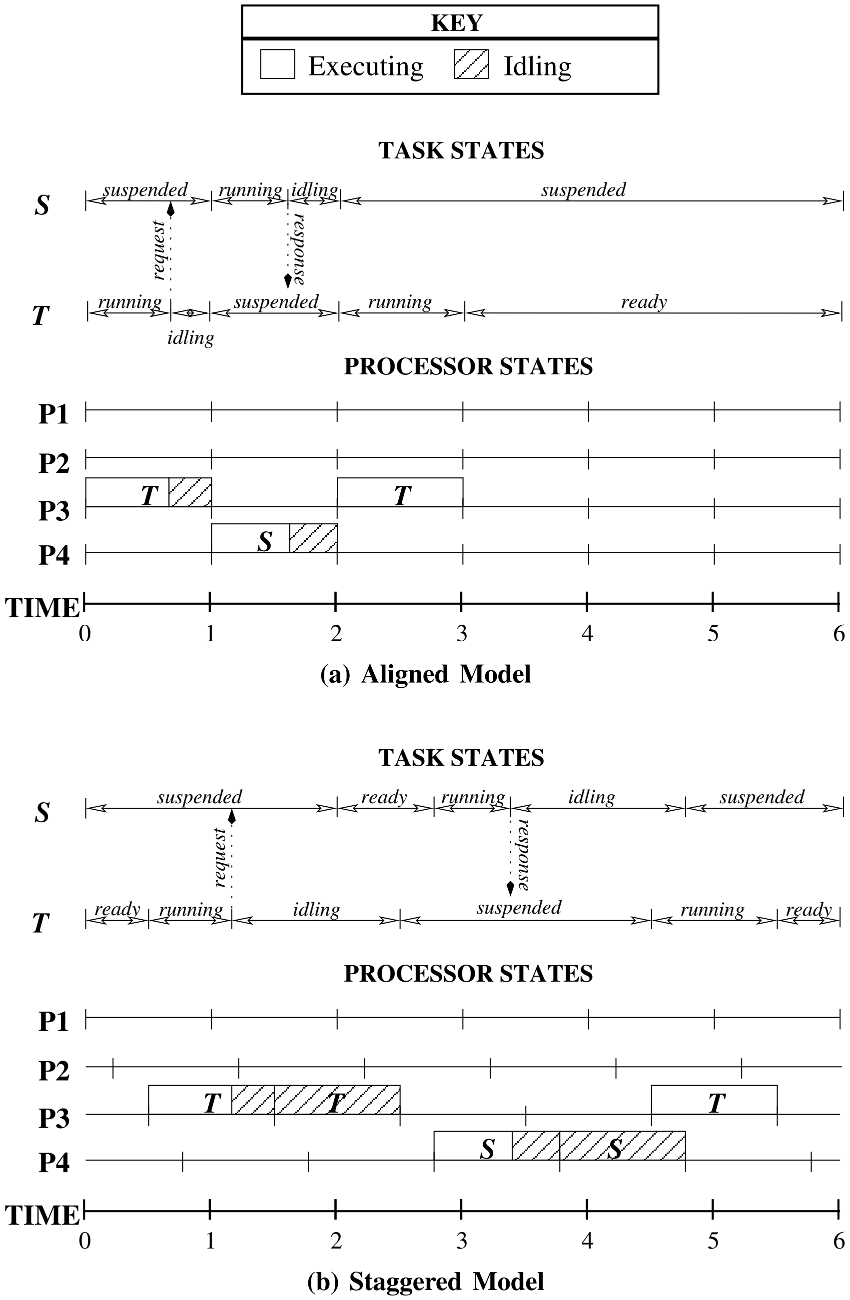


Fig. 9. Illustration of how staggering adversely impacts client/server designs. The servicing of one request is shown **(a)** under the aligned model and **(b)** under the staggered model.

E. Server-based Lock Synchronization

In [8], [11], a simple analysis is presented for a server-based locking protocol, called the Static-weight Server Protocol. Because this analysis is based on the assumption that every competing task has its critical-section request executed before that of the requesting task T, the worst-case time required to process T's request is unaffected by (E1) and (E2). However, since both the requesting task and lock server may suspend under this protocol, they can suffer from the suspension-related problems mentioned earlier in Section IV-A. For the server, this problem translates into slightly inflated worst-case response times.

F. Evaluating the Trade-off

Staggering represents a trade-off between off-line schedulability and on-line performance. Based on the above discussion, staggering introduces additional overhead in four areas: task weight assignment, supertask weight assignment, suspensions, and synchronization. We consider each below.

First, consider task weight assignment. Based on the observations made in Section IV-A, the increase in weight that is required to ensure strict deadlines for a task T is approximately $\frac{a}{b-1} - \frac{a}{b}$. This estimate simplifies to $\frac{a}{b(b-1)}$. It follows that mapping overhead is most impacted when b is small, *i.e.*, comparable to the slot size. However, quantum-based scheduling cannot efficiently support tasks with such tight constraints because many forms of overhead are prohibitively high in such cases, regardless of the scheduling model. (See [8] for a detailed discussion of this issue.) For such cases, alternative scheduling approaches must be considered. Hence, such cases should occur rarely, if at all, in systems using Pfair scheduling.

Second, consider supertask weight assignment using our reweighting approach [9], [12]. To estimate the impact of staggering on this aspect of Pfair-scheduled systems, we conducted a simple experiment. To facilitate the discussion, we omit the details of the experiment here. (The setup and results of the experiment are discussed in detail in [8].) The results suggest that staggering increases supertasking overhead by a factor of approximately 2.5. However, other experimental results (which can also be found in [8]) suggest that supertasking overhead is typically very low. Consequently, we expect supertasking overhead to remain reasonably low, despite the influence of staggering.

Third, consider the overhead introduced by suspensions. As explained earlier in Section I, a task that yields before the next scheduling point under quantum-based scheduling is still charged for the unused processor time. Hence, frequent suspensions lead to both poor task performance and low processor utilization. To achieve good performance, tasks exhibiting such behaviors should *not* be included in the global schedule, if it can be avoided. Indeed, this observation was a driving motivation behind our consideration of fully preemptive supertasking [12], which allows such tasks to be scheduled *as a group* rather than individually. The benefit of this approach is that the global scheduler must handle only the supertask, which never suspends. In addition, component tasks can be scheduled using a uniprocessor algorithm that is more flexible with respect to suspensions. Given the availability of mechanisms that avoid this problem, suspensions are expected to occur infrequently in the global schedule and, hence, should contribute little to the actual cost of staggering.

Finally, consider synchronization overhead. To optimize performance, synchronization at the global level should be limited to lock-free and zone-based locking synchronization, if possible. Prior experimental work [8] suggests that these approaches tend to produce very low overhead in most cases. Since staggering at most doubles this overhead, the cost of synchronization should remain reasonable under staggering.

Based on the above reasoning, staggering is expected to produce a reasonable increase in overhead in most systems. Indeed, it seems that staggering introduces significant overhead primarily in situations in which Pfair scheduling is likely to perform poorly anyway. More importantly, experimental results presented in the next section suggest that substantial performance gains can be achieved through staggering. These gains translate into decreased execution times, which are beneficial in both the average and worst cases. We expect the losses considered in this section to be exceeded by the suggested gains.

V. Experimental Results

In this section, we present the results of an experimental study of staggering and the proposed scheduling algorithm.

A. Simulations

In this subsection, we present a simulation-based comparison of the Pfair models. The advantage of simulation over direct measurement is that the processor count can be varied and the cache characteristics can be set arbitrarily. The latter trait was used to test performance on a simple blocking cache.

Experimental setup. These experiments used the Limes [14] simulator. Limes simulates execution close to the hardware level. As such, it provides no process management. However, Limes permits hardware-level aspects of the system to be more easily controlled than more complex simulators. Due to the limitations of Limes, preemption effects were only simulated. Specifically, caches were initially filled with dirty lines. Each task then performed writes to a local array until a "preemption" was detected. Tasks reacted to preemptions by exchanging their arrays for other uncached local ones. Hence, the cold-cache effect of preemptions was simulated using working-set changes.

We considered processor counts in the range $1, \ldots, 16$ and reported behavior across three slots. The system consisted of 200 MHz i86 processors using a 10 millisecond quantum. Caching consisted of blocking, direct-mapped caches with 256 KB capacity, 32-byte lines, and a MESI[6] coherency protocol [18]. Since our goal was to measure only preemption-related contention, both actual and false data sharing was avoided.[7]

Bus contention was measured by counting pending bus requests at each cycle. Since only one request could be serviced in each cycle, the presence of i requests implied that $i - 1$ tasks wasted that cycle waiting for bus access. Cycles required to service a task's bus requests were *not* counted as wasted cycles.

Relevance. These simulations focused on simultaneously scheduled tasks that process large data sets. Consideration of this scenario is motivated by the fact that many real-time multiprocessor systems consist of significant numbers of such tasks. Signal-processing and virtual-reality systems are two examples. Hence, we believe that these scenarios do represent situations that can arise in practice.

Worst-case contention under the aligned model. The first experiment estimated the worst-case bus contention under the aligned model (when working sets are fully cacheable). (This experiment does *not* characterize the worst case for staggering.) The worst case occurs when each task writes to each cache line in its working set at the start of each quantum. Figure 11 shows the average number of cycles lost per task.

Notice that both curves converge as the processor count increases; this indicates an overload of the bus. When tasks fail to completely load their working sets within a single quantum, the resulting traffic pattern is approximately uniform across every quantum. As a result, staggering provides no benefit.

[6]MESI is an acronym based on the four possible states of a cache line: modified, exclusive, shared, and invalid.

[7]False sharing occurs when two or more tasks access a common cache line, but do not share any data.

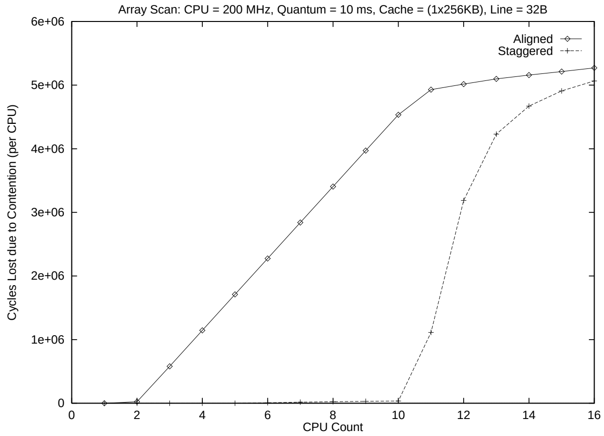
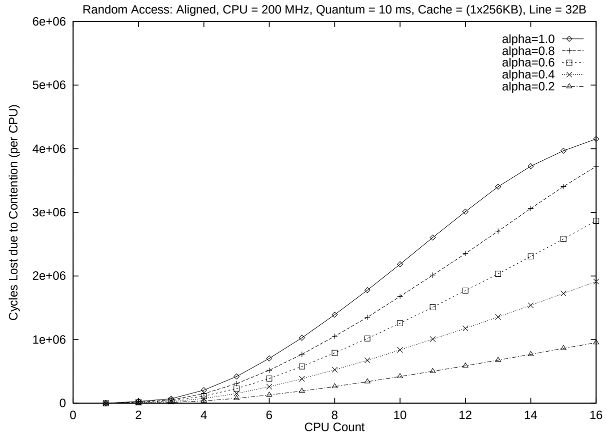


Fig. 11. Cycles lost to bus contention under each of the aligned and staggered models. Each working set completely fills the cache and lines are systematically written.

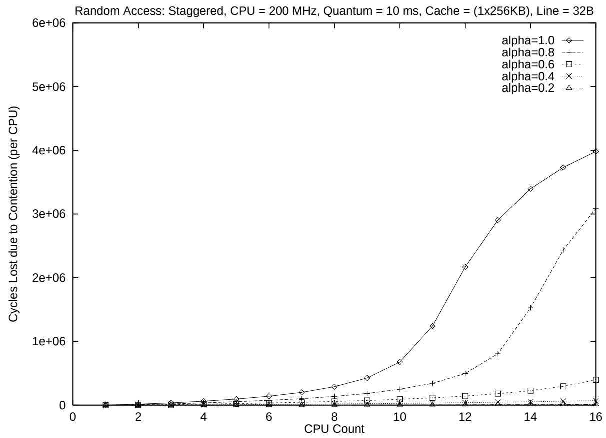
Hence, increasing the volume of bus traffic must eventually cause performance under both models to converge.

Random-access contention. Our second experiment measured performance under a random-access pattern when working-set sizes vary. Each task randomly selected cells to access from a fraction α of the full array. This behavior results in a burst of bus traffic at the start of each slot, followed by a gradual decline as the probability of referencing an uncached line decreases. The value of α was chosen from $\{\ 0.2k \mid 1 \leq k \leq 5\ \}$.

Results are shown in Figure 12. As shown, staggering produces significantly less loss in all cases. This experiment was repeated several times, producing virtually identical results. (These simulations were unfortunately too long to produce confidence intervals.)



(a) Aligned Model



(b) Staggered Model

Fig. 12. Cycles lost to bus contention under each of the **(a)** aligned and **(b)** staggered models. Random writes are performed and working-set sizes are varied.

B. Prototype Measurements

In this subsection, we present a comparison of the staggered and aligned models using a simple prototype of a Pfair system.

Experimental setup. The prototype microkernel executes as a thread package within QNX Neutrino 6.2.[8] The system consisted of four 200 MHz Pentium Pro processors, each of which had a 4-way, 512 KB L2 cache. These processors provide several latency-hiding features, including out-of-order execution, branch prediction, non-blocking caches, and support for multiple pending bus operations. Hence, this experiment will demonstrate whether staggering can improve upon simply applying common hardware-based techniques. Staggering should provide a much greater benefit to systems with fewer latency-hiding features.

Both experiments described in the previous section were conducted on the prototype. Due to the hardware complexity, performance was measured at the user level by calculating the average number of cycles per write operation in each quantum.

Results. Figure 13 shows the average number of cycles per write under the linear-access and random-access reference patterns. 99% confidence intervals were computed, but are omitted due to scale. (Marked intervals show the observed sample range.) As shown, staggering provides an increasing improvement until the array size reaches approximately 150 KB, at which point overload occurs.

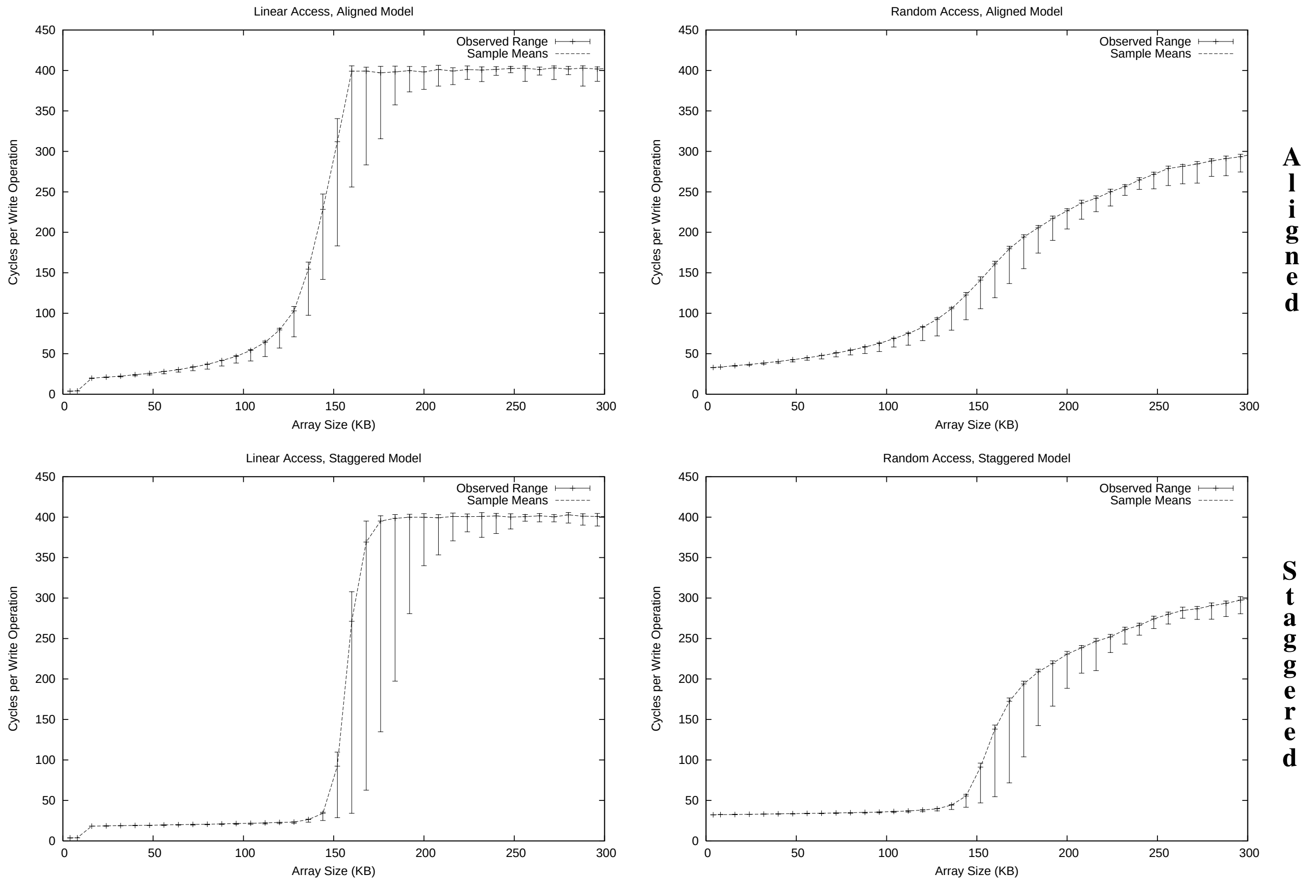
Figure 14 shows the ratios of corresponding sample means from the previous graphs. As shown, up to 7 (respectively, 2.5) times more writes were performed under the staggered model with the linear-access (respectively, random-access) pattern. Recall that this comparison is on a platform *with* latency-hiding features: improvement should be more dramatic without such features.

C. Scheduling Overhead

In the final series of experiments, the per-slot scheduling overhead of our staggered algorithm was compared to that of the master/slave PD^2 algorithm from [1].

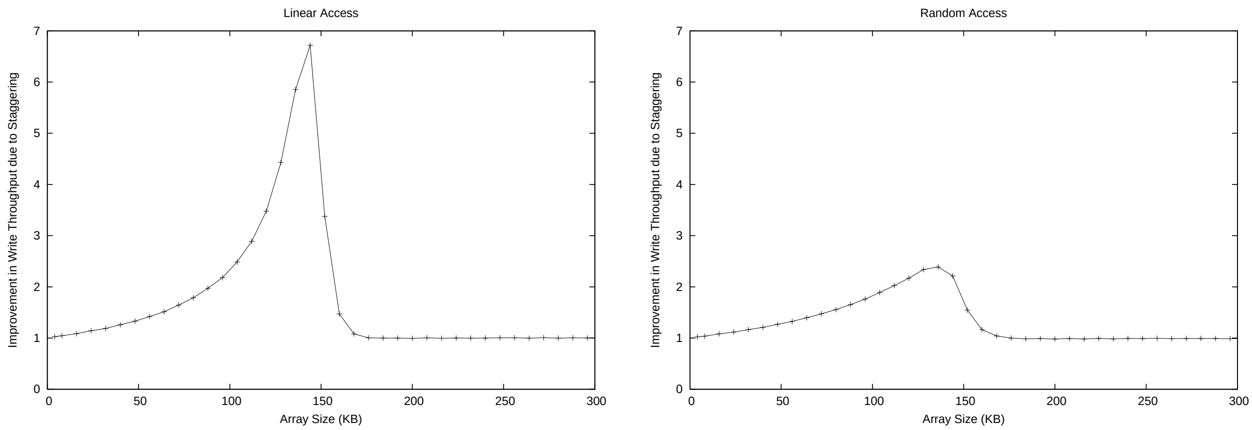
Experimental setup. Each experiment tested 1,000 randomly generated sets of independent tasks for each pairing

[8]The prototype takes control of the system when running. The underlying kernel activates only to process timer interrupts and to generate the signals that drive the prototype kernel. Neutrino was selected specifically to support this design.



Linear Access **Random Access**

Fig. 13. Results of linear- (left) and random-access (right) experiments conducted in the prototype Pfair system.

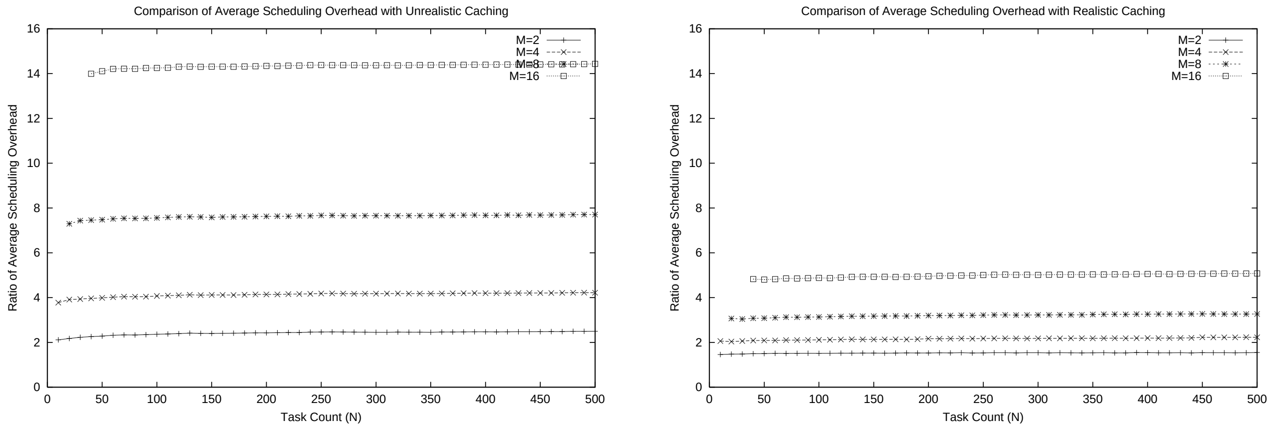


(a) Linear Access **(b) Random Access**

Fig. 14. Ratio of write throughput measured under the aligned model to that measured under the staggered model using the prototype Pfair system.

of $N \in \{\ 10n \mid 1 \leq n \leq 50\ \}$ and $M \in \{2, 4, 8, 16\}$. From the execution-time measurements, the ratio of the average per-slot overhead of the master/slave algorithm to that of the staggered algorithm was computed. Again, 99% confidence intervals were computed, but are omitted due to scale.

Warm cache. In the first experiment, we considered performance when scheduler invocations are performed iteratively on a uniprocessor (a 700-MHz Dell PC running Red Hat Linux 2.4). After a warm-up delay, all memory references hit in cache. This experiment approximates the best-case performance of each algorithm. Architectures with highly effective latency-hiding features should provide comparable performance on average. Figure 15(a) shows the results from this experiment. As shown, the staggered algorithm approaches, and often matches, the factor-of-M improvement suggested by its time complexity.



(a) Ideal Latency Hiding **(b) Realistic Caching**

Fig. 15. Ratio of the average per-slot scheduling overhead of master/slave scheduling to that of staggered scheduling. Measurements reflect performance **(a)** when all memory latency is hidden, and **(b)** when realistic caching is assumed.

Cold cache. Due to idealized cache behavior, the previous experiment provides no insight into how the algorithms perform on simpler architectures or in worst-case situations. To provide more realistic estimates, we performed a second comparison on an SGI Reality Monster with 32 300-MHz R10000 processors in which multiple copies of the scheduler were distributed on the processors. Control was then transfered between these copies at appropriate points so that the scheduler would encounter a (somewhat) cold cache.[9] Under master/slave (respectively, staggered) scheduling, these transfers occurred after each scheduler (respectively, `Schedule`) invocation. Figure 15(b) shows the results from this experiment. Caching effects close the performance gap substantially compared to the previous experiment. Despite this, staggering still provides a significant improvement. Indeed, since each invocation of `Schedule` makes fewer memory references than the master/slave algorithm, staggered scheduling should *never* produce more overhead, regardless of the platform. However, the magnitude of the improvement may vary significantly, as these experiments suggest.

VI. Conclusion

Although SMPs are well-suited to Pfair scheduling in many ways, experimental results presented herein suggest that preemption-related bus contention can significantly degrade performance. To address this problem, we proposed and demonstrated the effectiveness of a staggered model under which preemption-related bus traffic is more evenly distributed over time. Furthermore, we developed and experimentally evaluated an efficient scheduling algorithm to support this model that also produces less scheduling overhead than current Pfair algorithms. Finally, we explained how existing results can be applied, with minor modification, to the proposed model. In future work, we intend to extend our Pfair prototype to enable an empirical comparison to the partitioning approach.

[9]There is no way to accurately predict the cache states that a scheduler will encounter in a real system. We chose this approach because it seemed reasonable and straightforward to implement.

As the discussion in Section IV illustrated, there are many advantages to using aligned quanta. Most notably, a task executes in parallel with at most $M-1$ other tasks. Such a guarantee is particularly helpful when bounding the worst-case contention for shared resources. Because staggering sacrifices these advantages, it must inevitably suffer some negative side effects. However, as explained in Section IV, it appears that these side effects most impact cases in which Pfair and similar techniques are likely to perform poorly anyway. For the remaining cases, the loss appears to be modest and strongly offset by the potential performance gains demonstrated by the experiments described in Section V. However, those gains are a function of the contention for the bus, and hence of the amount of data used by tasks. When tasks primarily use very small data sets, the negative side effects of staggering will likely outweight the benefits.

Acknowledgement: We are grateful to the anonymous reviewers for their helpful suggestions regarding an earlier draft of this paper.

References

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

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

[3] B. Andersson and J. Jonsson. Fixed-priority preemtive multiprocessor scheduling: to partition or not to partition. In *Proceedings of the Seventh International Conference on Real-time Computing Systems and Applications*, pages 337–346, December 2000.

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

[6] A. Chandra, M. Adler, and P. Shenoy. Deadline fair scheduling: bridging the theory and practice of proportionate fair scheduling in multiprocessor systems. In *Proceedings of the 7th IEEE Real-time Technology and Applications Symposium*, May 2001.

[7] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman. 1979.

[8] P. Holman. On the implementation of Pfair-scheduled multiprocessor systems. Ph.D. Thesis, University of North Carolina at Chapel Hill. 2004.

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

[10] P. Holman and J. Anderson. Object sharing in Pfair-scheduled multiprocessor systems. In *Proceedings of the 14th Euromicro Conference on Real-time Systems*, pp. 111-120, June 2002.

[11] P. Holman and J. Anderson. Locking in Pfair-scheduled multiprocessor systems. In *Proceedings of the 23rd IEEE Real-time Systems Symposium*, pages 149–158, December 2002.

[12] P. Holman and J. Anderson. Using hierarchal scheduling to improve resource utilization in multiprocessor real-time systems. In *Proceedings of the 15th Euromicro Conference on Real-time Systems*, pages 41–50, July 2003.

[13] P. Holman and J. Anderson. Implementing Pfairness on a symmetric multiprocessor. In *Proceedings of the 10th IEEE Real-time Technology and Applications Symposium*, pages 544–553, May 2004.

[14] I. Ikodinovic, D. Magdic, A. Milenkovic, and V. Milutinovic. Limes: a multiprocessor simulation environment for PC platforms. In *Proceedings of the 3rd International Conference on Parallel Processing and Applied Mathematics*, pp. 398–412, September 1999.

[15] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real–time environment. *Journal of the ACM*, 30:46–61, January 1973.

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

[17] A. Mok. Fundamental design problems for the hard real-time environment. Ph.D. Thesis, Massachussetts Institute of Technology. 1983.

[18] M. Papamarcos and J. Patel. A low overhead solution for multiprocessors with private cache memories. In *Proceedings of the 11th International Symposium on Computer Architecture*, pages 348–354, June 1984.

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

[20] A. Srinivasan and J. Anderson. Fair scheduling of dynamic task systems on multiprocessors. Presented at the *11th International Workshop on Parallel and Distributed Real-time Systems* (on CD-ROM), April 2003.

[21] A. Srinivasan, P. Holman, J. Anderson, and S. Baruah. The case for fair multiprocessor scheduling. Presented at the *11th International Workshop on Parallel and Distributed Real-time Systems* (on CD-ROM), April 2003.

[22] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the Seventh IEEE Real-time Systems Symposium*, pp. 288–299, December 1996.