Implementing Pfairness on a Symmetric Multiprocessor*

Philip Holman and James H. Anderson

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
E-mail: {holman,anderson}@cs.unc.edu
January 2004

Abstract

We consider the implementation of a Pfair 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.

*Work supported by NSF grants CCR 9988327, ITR 0082866, CCR 0204312, and CCR 0309825, and by a grant of software provided by the QNX corporation.

1 Introduction

Recent research on real-time multiprocessor scheduling has shown that global scheduling provides many advantages over partitioning approaches, including improved schedulability and flexibility [1, 2, 12]. Furthermore, these benefits can be achieved without incurring significantly more (worst-case) overhead [14]. 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. 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 [3] is a particularly promising global-scheduling approach. Indeed, Pfair scheduling is presently the only known optimal means for scheduling periodic, sporadic, and rate-based tasks on a multiprocessor [12]. Hence, Pfair scheduling is seemingly well-suited for systems in which worst-case predictability is required. However, some aspects of Pfair scheduling may degrade performance and have led to questions regarding its practicality. These aspects are discussed in detail below.

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

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 [7, 8], techniques for accounting for system overheads [14], and support for hierarchal scheduling [6, 7, 8, 9]. 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 overhead low. In addition, the worst-case cost of a migration is effectively the same as that of a preemption under a cache-based SMP.

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 for 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.

Fig. 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 it simply wrote to each cache line iteratively. As shown, heavy contention follows each scheduling point. Other results, presented later, suggest that such contention can significantly lengthen the execution times of tasks.

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 measure-

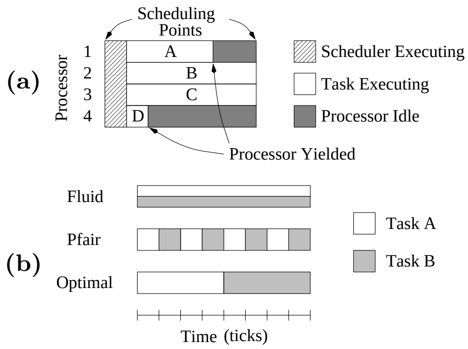
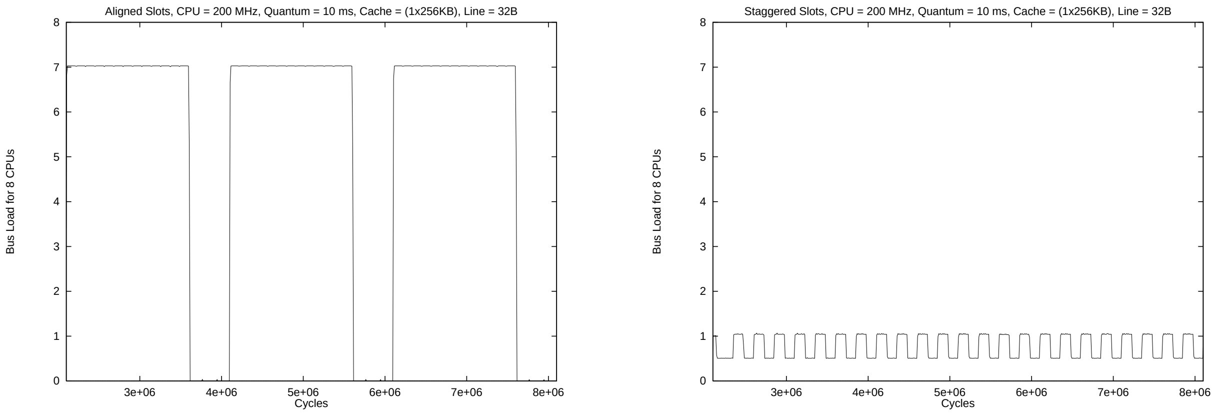


Figure 1: **(a)** Two scheduled tasks yield before the next scheduling point. **(b)** Relationship between a Pfair schedule (middle), the corresponding fluid schedule (upper), and a schedule that minimizes context switching (lower).

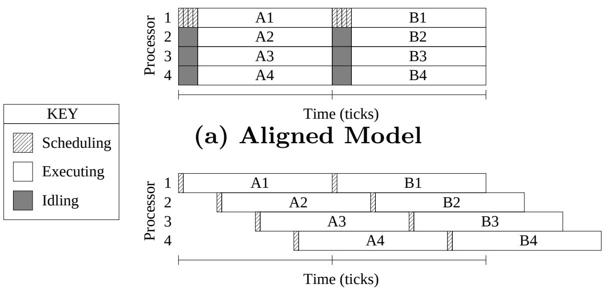


(a) Aligned Model **(b) Staggered Model**

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

ments 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.

Fig. 3(b) shows the proposed staggered model alongside the traditional *aligned* model. Repeating the earlier experiment using the staggered model results in dramatically lower contention, as shown in Fig. 2(b). Under the aligned model, all scheduling decisions are made by processor 1 (see Fig. 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 Fig. 3(b)). By avoiding processor stalls, both the worst-case and average-case scheduling overhead is reduced.



(a) Aligned Model

(b) Staggered Model

Figure 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.

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.) Without analysis, these solutions are not appropriate for hard real-time systems, which are the focus of our work. Techniques for optimizing Pfair scheduling for a general-purpose environment (again, without supporting analysis) were previously proposed by Chandra, Adler, and Shenoy [5].

The remainder of the paper is organized as follows. Sec. 2 summarizes Pfair scheduling. An efficient scheduling algorithm for the staggered model is presented in Sec. 3. In Sec. 4, we explain how prior work relates to the proposed model. Experimental results are then presented in Sec. 5. We conclude in Sec. 6.

2 Background

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

2.1 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. 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.

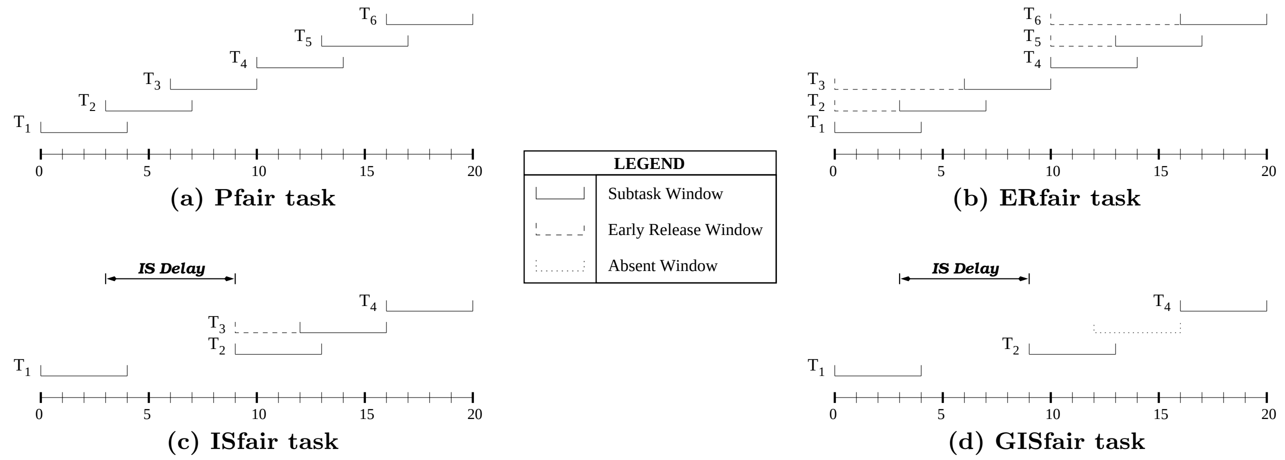


Figure 4: The windowing for a task with weight $T.w = 3/10$ is shown under a variety of circumstances. **(a)** Normal windowing used by a Pfair task. **(b)** Early releases have been added to the windowing in (a) so that each grouping of three subtasks becomes eligible simultaneously. (In reality, each subtask will not become eligible until its predecessor is scheduled.) **(c)** Windows appear as in (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 (c) except that T_3 is now absent.

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.

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. The concept of tracking the ideal system is formalized by the notion of *lag*. Letting $A(T, t)$ denote the number of quanta allocated to task T over the time interval $[0, t)$, the *lag* of T at t can be formally defined as $lag(T, t) = T.w \cdot t - A(T, t)$. A schedule respects Pfairness if and only if the magnitude of all task lags is strictly less than one always, *i.e.*, $(\forall T, t :: |lag(T, t)| < 1)$. As shown in [3], a schedule respecting Pfairness exists if and only if $\sum_{T \in \tau} T.w \leq M$.

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. Fig. 4(a) shows the time-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 said to be *released* at time 3 and to have a *deadline* at time 7.

At present, PF [3], PD [4], 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 $t(i, k) = i + \frac{k}{M}$.

2.2 Extensions

We now discuss relevant extensions to Pfair scheduling.

Increased flexibility. Srinivasan and Anderson [1, 12] 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 the processor count. *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. Fig. 4 illustrates these variants. In addition to these variants, Srinivasan and Anderson also derived conditions under which tasks may leave and join a system [13].

Supertasking. *Supertasks* were first proposed to support the static assignment of tasks to processors [11]. 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 algorithms for deriving supertask weights from component task sets using either quantum-based [6] or event-driven [9] scheduling. In addition to hierarchal-scheduling support, supertasks also 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, as demonstrated in [7, 8].

Synchronization. In other prior work, we developed techniques for supporting lock-free [7] and lock-based [8] synchronization under Pfair scheduling. For lock-based synchronization, we presented two approaches: timer-based and server-based protocols. Timer-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.

2.3 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 [14]. 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 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 as it is refined. Partitioning, on the other hand, is well-studied and hence is not likely to improve.

3 Scheduling Algorithm

In this section, we present an efficient scheduling algorithm for the staggered model.

Back-to-back scheduling. Under staggering, quanta from different slots can overlap, as the A3 and B1 quanta illustrated in Fig. 3(b) (see Sec. 1). Hence, the scheduler must ensure that tasks scheduled *back-to-back* (*i.e.*, in consecutive slots) are not granted overlapping quanta. Safety is ensured by employing an affinity-based assignment policy. Specifically, tasks can be migrated only after preemptions. Hence, a task that is scheduled back-to-back will execute on the same processor in both slots. This and similar policies can be used to improve cache performance as well. When used with supertasks, the benefits of such a policy can be substantial, *i.e.*, the cache performance of a group of EDF-scheduled component tasks in a heavily weighted supertask will resemble that of using EDF scheduling on a dedicated uniprocessor.

Concept. Consider scheduling slot t on the first processor after executing task T in slot $t-1$. 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 [12].) In addition, the decision must respect the guarantee described above if T is scheduled back-to-back. However, identifying all tasks scheduled in slot t is an $\Omega(M)$ operation.

The implication is that the tasks scheduled in slot t must be identified *before* invoking the scheduler at the start of slot t. To achieve this, our algorithm divides scheduling into two steps: **(i)** up to M tasks (if that many are eligible) are selected to execute in slot t and stored in t's *scheduling set*, and **(ii)** each processor later selects a task to execute from those in t's scheduling set. To ensure that t's scheduling set is known at the start of slot t, Step (i) is actually performed *one slot early* (*i.e.*, by the scheduler invocations in slot $t-1$).

3.1 Basic Algorithm

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

Data structures. When scheduling slot t, tasks scheduled in slot t (respectively, $t+1$) 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 $t+1$. All heaps are ordered according to task priorities. (The $\preceq$ and $\succeq$ relations, which define this ordering, are defined below.) In addition, the *Incoming*$[t]$ heap stores tasks that will not be eligible to execute until slot t. (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.

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

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

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

procedure Initialize()
1:  Eligible := Eligible ∪ Incoming[0];
2:  while |Eligible| > 0 ∧ |SchedNext| < M do
3:      SchedNext := SchedNext ∪ {Extract-Max(Eligible)}
    do

procedure SelectTask(T)
4:  if T ∈ S_t then
5:      ReschedNext := ReschedNext ∪ {T}
    else
6:      SchedNext := SchedNext ∪ {T}
    fi

procedure Schedule(p)
7:  if SchedCount = M then
8:      t := t + 1;
9:      Eligible := Eligible ∪ Incoming[t + 1];
10:     Swap(SchedNow, SchedNext);
11:     Swap(ReschedNow, ReschedNext)
    fi;
12: if Running[p] ∈ ReschedNow then
13:     T := Running[p];
14:     ReschedNow := ReschedNow / {T}
15: else if |SchedNow| > 0 then
16:     T := Extract-Min(SchedNow)
    else
17:     T := ⊥
    fi;
18: Running[p] := T;
19: if T ≠ ⊥ then
20:     UpdatePriority(T);
21:     if T.elig ≤ t + 1 then
22:         Eligible := Eligible ∪ {T}
        else
23:         Incoming[T.elig] := Incoming[T.elig] ∪ {T}
        fi
    fi;
24: if |Eligible| > 0 then
25:     SelectTask(Extract-Max(Eligible))
    fi;
26: SchedCount := (SchedCount mod M) + 1
```

Figure 5: Basic staggered scheduling algorithm.

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 lower than (respectively, lower than or equal to) priority ρ_2. We further assume that `UpdatePriority()` encapsulates the algorithm for updating *prio* and *elig*.

The remaining variables include two counters (t and *SchedCount*) and the *Running* array. t is the index of the current slot. *SchedCount* indicates the number of scheduler invocations that have been performed for slot t. Finally, the *Running* array indicates which task is currently executing on each processor.

To simplify presentation, some branches test for set inclusion ($\in$) and the branch at line 4 uses S_t to denote the set of tasks selected to execute in slot t. 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 and schedules slot 0. 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 $t+1$. Line 4 determines whether T is scheduled back-to-back. T is then stored in the proper heap in lines 5–6.

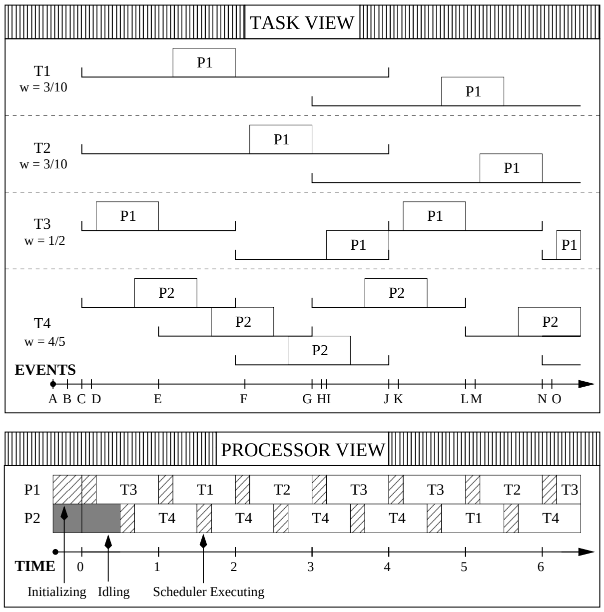
`Schedule`(p) is invoked to schedule processor p. Lines 8–11 initialize a round of scheduler invocations by incrementing t, by merging newly eligible tasks into *Eligible*, and by initializing *SchedNow*, *ReschedNow*, *SchedNext*, and *ReschedNext*. Lines 12–18 select the task to execute on processor 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 $t+1$. Finally, progress is recorded by updating *SchedCount* at line 26.

Example. Fig. 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 presented in the right frame of Fig. 6.

3.2 Extensions

We now present the procedures needed to support tasks leaving and joining the system (shown in Fig. 7).

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. Second, adding and removing tasks can lead to incorrect scheduling deci-



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).

Figure 6: Four tasks (T1–T4) are executed on two processors (P1–P2) under the staggered model. The left frames show two views of the schedule. The right frame shows a trace of task T3's movement through the scheduler's heaps. Event labels in the trace correspond to the event timeline in the upper-left frame.

sions. To avoid this potential problem, the presented procedures ensures 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 $T \in (\mathit{SchedNext} \cup \mathit{ReschedNext})$ and $U \in \mathit{Eligible}$, $T.\mathit{prio} \succeq U.\mathit{prio}$

(I2) $|\mathit{SchedNext}| + |\mathit{ReschedNext}| < \mathit{SchedCount}$ and $|\mathit{Eligible}| = 0$

Detailed description. Invoking `Deactivate`(T) in slot t causes task T to be ignored when scheduling slots at and after $t + 1$. If T has been selected to execute in slot t 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: **(i)** T is scheduled in slot $t + 1$ (it is in either *SchedNext* or *ReschedNext*), **(ii)** T is scheduled in slot t but has not been granted a processor (it is in either *SchedNow* or *ReschedNow*), and **(iii)** T is not scheduled in slot $t+1$ (but may be currently executing in slot t). 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 t causes task T to be considered when scheduling slots at and after $t + 1$. Again, three cases must be considered: **(i)** T is not eligible to execute in slot $t+1$, **(ii)** T is eligible to execute in slot $t + 1$ and a processor will idle in that slot, and **(iii)** T is eligible to execute in slot $t+1$ 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 $t + 1$. 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 $t + 1$ is then stored in *Eligible* at line 50.

3.3 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

```
procedure Deactivate(T)
if T ∈ SchedNext ∪ ReschedNext then
    if T ∈ ReschedNext then
        ReschedNext := ReschedNext / {T}
    else
        SchedNext := SchedNext / {T}
    fi;
    if |Eligible| > 0 then
        SelectTask(Extract-Max(Eligible))
    fi
else if T ∈ SchedNow ∪ ReschedNow then
    if T ∈ ReschedNow then
        ReschedNow := ReschedNow / {T}
    else
        SchedNow := SchedNow / {T}
    fi;
    UpdatePriority(T)
else if T ∈ Eligible then
    Eligible := Eligible / {T}
else
    Incoming[T.elig] := Incoming[T.elig] / {T}
fi
```

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

procedure Activate(T)
if T.elig ≤ t + 1 then
    if |SchedNext| + |ReschedNext| < SchedCount then
        SelectTask(T)
    else
        if |SchedNext| = 0 ∨ (|ReschedNext| > 0
            ∧ Min(SchedNext).prio ⪰ Min(ReschedNext).prio) then
            H := &ReschedNext
        else
            H := &SchedNext
        fi;
        if Min(*H).prio ≺ T.prio then
            SelectTask(T);
            T := Extract-Min(*H)
        fi;
        Eligible := Eligible ∪ {T}
    fi
else
    Incoming[T.elig] := Incoming[T.elig] ∪ {T}
fi
```

Figure 7: Extensions to support departures and suspensions (left), and arrivals and resumptions (right).

$O(M \log N)$ 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.)

4 Impact on Analysis

We now discuss how staggering impacts prior results.

Side effects. A staggered slot extends up to $\Delta = \frac{M-1}{M}$ beyond its placement when aligned. Hence, slot i on a processor may overlap slots $i+1$ and $i-1$ on other processors, which leads to the following side effects.

(E1) An event occurring at t under the aligned model may be delayed until $t + \Delta$ under staggering.

(E2) Each quantum overlaps $M-1$ other quanta when aligned, but $2(M-1)$ when staggered.

Basic analysis. Consider independent-task scheduling. Independent tasks are oblivious to the execution of other tasks and, hence, are unaffected by (E2). (E1), on the other hand, has two implications.

First, deadlines guaranteed under the aligned model may be missed by up to Δ under staggering. Deadlines can be strictly enforced by increasing each task's weight so that each strict deadline occurs one slot earlier than its true position. 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.

Second, events (such as suspension requests) occurring in slot t may occur *after* time $t+1$, at which point the scheduling decisions for slot $t+1$ are committed. As a result, a suspending task may occasionally (when scheduled back-to-back) cause a quantum to be wasted, as illustrated in Fig. 8. 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.

Supertasking. In [6, 9], a supertask weight is cho-

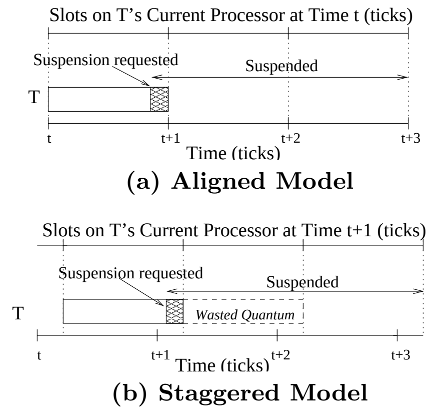


Figure 8: Illustration of how an event timing differs between **(a)** the aligned model and **(b)** the staggered model. An event in slot t is no longer guaranteed to occur before time $t+1$ under the staggered model. Due to this, a task may have already been scheduled in the next slot before a suspension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

sen by comparing the least amount of processor time guaranteed to a supertask to the total processor-time requirement of all component tasks. Only the first of these two quantities requires adjustment due to staggering, and is independent of (E2).

The primary impact of (E1) is that the amount of processor time guaranteed to the supertask is slightly lower under staggering. As in the case of independent tasks, this delay may result in bounded (component-task) deadline misses. Strict deadlines can be guaranteed by simply applying the analysis presented in [9] with an adjusted estimate of the amount of processor time granted to the supertask.

Lock-free synchronization. Lock-free operations are optimistically attempted and retried until successful. A bound on retry overhead can be derived by determining the worst-case number of failures. The analysis presented in [7] assumes that **(i)** a failure implies the success of a competing operation, and **(ii)** each operation is short enough to be preempted at most once. By (i), the number of failures experienced by some task T due to parallel interference within a slot is bounded by the number of operations that can be performed in parallel within that slot. Hence, under the aligned model, it is sufficient to assume that the worst-case mix of $M-1$ interfering tasks executes in parallel with T. By (E2), similar reasoning can still be applied under staggering; however, $2(M-1)$ tasks must now be considered, which potentially doubles the overhead.

Timer-based lock synchronization. Two timer-based locking protocols are presented in [8]. Each delays lock-requesting operations that are at risk of being preempted until the start of a new quantum. (See [8] for details.) This action prevents preemptions from lengthening the duration of time that a lock is held. Both protocols guarantee that only executing tasks can hold locks, and hence can block a requesting task T. As in the lock-free case, this analysis assumes that each task T can be preempted at most once before an operation completes. Hence, the situation is quite similar to that discussed above: bounds on 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.

Server-based lock synchronization. A server-based approach that uses a FIFO-ordered request queue is also presented in [8]. Since the analysis assumes that every competing task has its request executed before 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 may suffer from the suspension-related problems mentioned earlier.

Trade-off. Staggering represents a trade-off between off-line schedulability and on-line performance. Of the costs mentioned above, those related to suspensions and synchronization will likely be the highest in practice. We consider each below.

As mentioned in Sec. 1, a task that yields before the next scheduling point is charged for the unused processor time. Hence, frequent suspensions will likely lead to both poor task performance and low utilization. To achieve good performance, tasks exhibiting such behaviors should *not* be Pfair-scheduled. Indeed, this observation was a driving motivation behind our work on supertasking [9], which allows such tasks to be scheduled *as a group* rather than individually. The benefit is that the global scheduler sees only the supertask, which never suspends, while component tasks can be scheduled using a uniprocessor algorithm that is more flexible (with respect to suspensions). Given the availability of mechanisms to help avoid this problem, suspensions are expected to occur infrequently in the global schedule and, hence, should contribute little to the actual cost of staggering.

Similarly, synchronization at the global level should be limited to lock-free and timer-based locking synchronization. As shown in [7, 8], these approaches tend to result in very low overhead in practice. Since staggering at most doubles this overhead, the cost of synchronization should remain relatively low under staggering.

Based on this reasoning, staggering is expected to result in only slight schedulability loss in practice. 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 case. We expect the losses considered in this section to be exceeded by the gains provided by these decreased execution requirements.

5 Experimental Results

This section contains an experimental evaluation of staggering and the proposed scheduling algorithm.

5.1 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 [10] 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 consider processor counts in the range $[1, 16]$ and report 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 and 32-byte lines. The MESI coherency protocol was used. Since our goal was to measure only preemption-related contention, both actual and false data sharing was avoided.

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 k requests implied that $k-1$ 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 focus 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. Fig. 9(a) shows the average number of cycles lost per task.

Notice that both curves converge as the processor count increases, which 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. 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 selects 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 Fig. 9(b)–(c). 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.)

5.2 Prototype Measurements

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

Experimental setup. The prototype microkernel executes as a thread package within QNX Neutrino 6.2.[1] The system consists of four 200 MHz Pentium Pro processors, each of which has 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. Results are shown in Fig. 10. The upper two graphs in the left (respectively, right) column show the average number of cycles per write under the linear-access (respectively, random-access) reference pattern. 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.

The two graphs at the bottom of Fig. 10 show the ratios of corresponding sample means from the other 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.

[1]The prototype takes control of the system when running. Neutrino was selected specifically to support this approach.

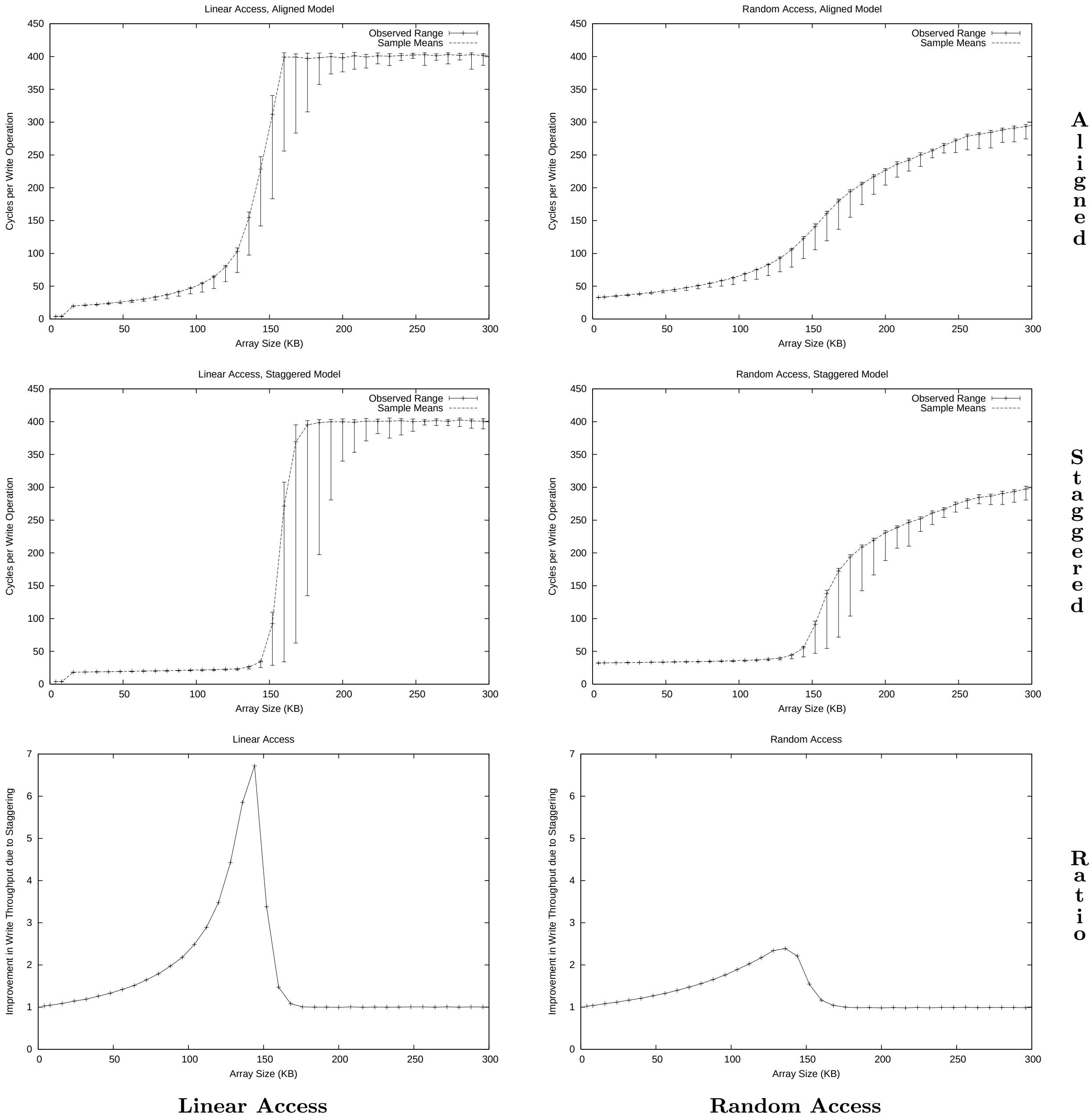


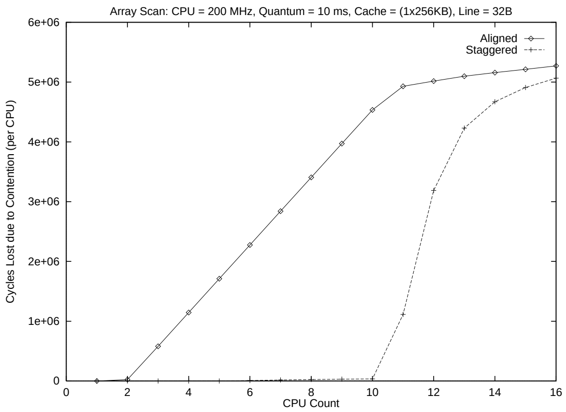
Figure 10: Results of linear- (left) and random-access (right) experiments conducted in the prototype Pfair system.

5.3 Scheduling Overhead

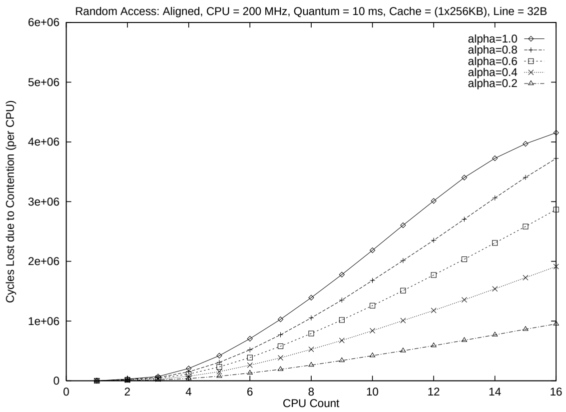
In the final series of experiments, the per-slot scheduling overhead of the algorithm from Sec. 3 was compared to that of the master/slave PD^2 algorithm from [1].

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

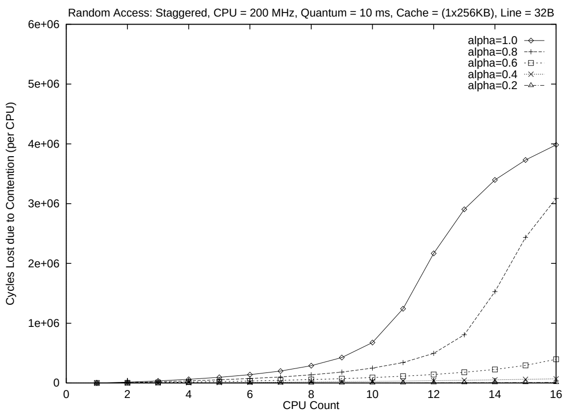
Warm cache. In the first experiment, we considered performance when scheduler invocations are performed iteratively on a uniprocessor (a 700-MHz Dell PC running Red Hat Linux 2.4). After a warm-up delay, all memory references hit in cache. This experiment approximates the best-case performance of each algorithm. Architectures with highly effective latency-hiding features should provide comparable performance on average. Fig. 11(a) shows the results from this



(a) Worst Case for Aligned Model



(b) Random Access: Aligned Model



(c) Random Access: Staggered Model

Figure 9: Cycles lost to bus contention are shown. **(a)** Each working set completely fills the cache and lines are systematically written. **(b)**–**(c)** Random writes are performed and working-set sizes are varied.

experiment. As shown, the staggered algorithm approaches, and often matches, the factor-of-M improvement suggested by its time complexity.

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.[2] Under master/slave (respectively, staggered) scheduling, these transfers occurred after each scheduler (respectively, `Schedule`) invocation. Fig. 11(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.

6 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.

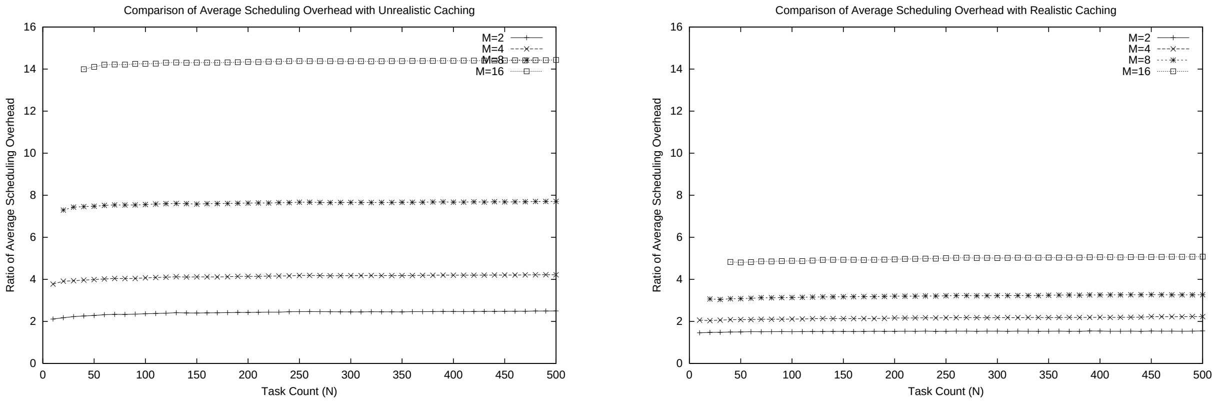
Acknowledgement: The authors would like to thank Prashant Shenoy for reading an earlier draft of this paper and suggesting ways to improve the presentation.

References

[1] J. Anderson and A. Srinivasan. Early-release fair scheduling. In *Proceedings of the 12th Euromicro Conference on Real-time Systems*, pp. 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*, pp. 76–85, June 2001.

[2] 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.



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

Figure 11: 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.

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

[4] 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*, pp. 280–288, April 1995.

[5] 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.

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

[7] 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.

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

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

[10] 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.

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

[12] 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.

[13] 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*, April 2003.

[14] 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*, April 2003.