The Staggered Model:
Improving the Practicality of Pfair Scheduling*

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
May 2003

Abstract

We consider the implementation of a Pfair scheduler on a symmetric multiprocessor (SMP) architecture. Although SMPs seem well-suited for Pfair scheduling, simulations 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 under which scheduling points are uniformly distributed over time. Additional simulations are presented to demonstrate the effectiveness of the staggered model at reducing bus contention. To facilitate the use of the staggered model, we present an efficient scheduling algorithm and explain how existing Pfair results can be adapted for use under the new model. In addition to supporting the staggered model, the presented algorithm provides other practical benefits, including improved cache performance and reduced scheduling overhead.

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

Recent research on real-time multiprocessor scheduling has shown that global scheduling has many advantages over partitioning approaches, including improved schedulability and flexibility [1, 2, 12]. Despite these advantages, partitioning still provides practical advantages over global scheduling, including the use of well-understood uniprocessor techniques and improved cache performance. However, the most compelling advantage of partitioning is that it has been shown to be effective in real systems. Until global scheduling is shown to be capable of providing comparable performance in a real system, partitioning will continue to be the favored approach.

Proportionate-fair (Pfair) scheduling [3] is a particularly promising global-scheduling approach. Pfair scheduling is presently the only known optimal means for scheduling periodic, sporadic, and rate-based tasks on a multiprocessor [12]. Unfortunately, some aspects of Pfair scheduling have led to questions regarding its practicality.

One problematic aspect of Pfair scheduling is its use of synchronized tick scheduling. Specifically, 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. This is illustrated in Fig. 1(a).

A second problematic aspect of Pfair scheduling is its use of 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 tracking achieves theoretical optimality, but also tends to increase the number of context switches. The latter effect is undesirable since it increases scheduling overhead and reduces cache performance. Fig. 1(b) illustrates the relationship between a Pfair schedule, the corresponding fluid schedule, and a schedule that minimizes context switching.

A third potential concern is that task migration is unrestricted under Pfair scheduling. In the worst case, a task may be migrated each time it is scheduled, leading to reduced cache performance.

Contributions of this paper. In recent work, we have investigated methods of improving the efficiency of Pfair scheduling algorithms. Specifically, we have considered the problems of task synchronization [7, 8], accounting for practical overheads [14], and reducing overhead (including waste due to partially used quanta) via hierarchal

![Figure 1](image-url): (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).
Figure 2: Graphs show 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.

In this paper, we turn our attention to the problem of efficiently implementing a Pfair scheduler on a bus-based symmetric multiprocessor (SMP).

We consider a bus-based SMP for three reasons. First, symmetric processing is inherent under Pfair scheduling, as each task is assumed to be able to execute on any processor. Second, the frequency of context switches and migrations under Pfair scheduling suggests that tightly coupled processing is needed to keep these overheads reasonable. Third, the use of a central shared memory with coherent caches provides an efficient and transparent mechanism for migrating data between processors. In addition, the cost of a migration under such a memory system is effectively no different than the cost of a preemption.

Unfortunately, some aspects of Pfair scheduling do not favor a shared-bus architecture. In particular, a preempted task may find a cold cache that must be refilled when it resumes. Since refilling the cache leads to a period of increased bus traffic, scheduling all processors simultaneously can result in periodic bursts of very heavy bus contention. The worst-case duration of these bursts is then a function of the processor count and of the size of each task’s working set.

Fig. 2(a) illustrates this problem on eight processors. This figure shows the number of pending bus requests over an interval containing three scheduling points (i.e., spanning three quanta). To simulate worst-case behavior, each task uses an array that completely fills the cache and immediately writes to each cache line after being scheduled. As shown, heavy contention follows each scheduling point. Other simulation results, presented later, suggest that allowing such contention will lead to significantly longer task execution times.

To correct this problem, we propose a new model for Pfair scheduling in which scheduling points are uniformly
Figure 3: (a) Under the aligned (current) model, each processor is simultaneously scheduled and all scheduling decisions are made on a single processor. (b) Under the staggered (proposed) model, scheduling points are uniformly distributed over time, which both reduces bus contention and allows each processor to make its own scheduling decision.

distributed, but remain periodic on each processor. The proposed staggered model is shown in Fig. 3(b). For comparison, the current aligned model is shown in Fig. 3(a). To illustrate the benefit of the proposed model, Fig. 2(b) shows the result of repeating the earlier experiment using the new model. As shown, contention is more uniformly distributed and dramatically lower under the staggered model, as expected.

An additional benefit of staggering is that scheduling overhead can be efficiently distributed across processors. Specifically, existing scheduler implementations assume that all scheduling decisions are made by a single processor, as shown in Fig. 3(a). Under staggering, each processor can make its own scheduling decision without the need for explicit synchronization, as shown in Fig. 3(b). Because processor stalls are avoided when making scheduling decisions, scheduling overhead is reduced.

The remainder of the paper is organized as follows. Sec. 2 provides an overview of Pfair scheduling. To facilitate the use of the staggered model, we present an efficient scheduling algorithm in Sec. 3 and explain how existing results can be adapted for use under the new model in Sec. 4. We then present experimental results in Sec. 5, followed by a brief discussion of related work in Sec. 6. We conclude in Sec. 7.

2 Background

In this section, Pfair scheduling is formally defined and previous work in this area is summarized to provide an overview of the basic theory and its proposed extensions.

2.1 Basics of Pfair Scheduling

Let \( \tau \) denote a set of \( N \) tasks to be scheduled on \( M \) processors. Under Pfair scheduling, each task \( T \in \tau \) has a rational weight \( T.w \) in the range \((0,1]\). Conceptually, the weight of a task \( T \) is the rate at which \( T \) should be executed, relative to the speed of a single processor. Under the model of Baruah, Cohen, Plaxton, and Varvel [3], \( \tau \) is statically defined and tasks are assumed to be ready at all times, i.e., tasks never block or suspend. (This model was primarily intended to support independent periodic tasks.)
Pfair scheduling algorithms allocate processor time in discrete time units called quanta. The interval \([t, t+1)\), where \(t\) is a non-negative integer, is called time slot \(t\). In each time 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 the amount of processor time received by each task to the amount that would have been received in an ideal (fluid) system. Ideally, a task \(T\) with weight \(T.w\) 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 \(\text{lag}(T, t) = T.w \cdot t - A(T, t)\). A schedule respects Pfairness if and only if the magnitude of each task’s lag is strictly less than one at all times, i.e., \((\forall T, t :: |\text{lag}(T, t)| < 1)\).

Baruah et al. [3] showed that a schedule that respects Pfairness exists for a task set \(\tau\) on \(M\) processors if and only if the following condition holds.

\[
\sum_{T \in \tau} T.w \leq M
\]

(1)

The Pfairness 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 within which each subtask must execute to satisfy the Pfairness constraint when \(T.w = 3/10\). The interval associated with each subtask \(T_i\) is called \(T_i\)’s window. For example, the window of \(T_2\) in Fig. 4(a) is \([3, 7)\). Subtask \(T_2\) is said to be released at time 3 and to have a deadline at time 7.

At present, three optimal Pfair scheduling algorithms are known: PF [3], PD [4], and PD² [2]. Each

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.
prioritizes subtasks according to an earliest-subtask-deadline-first rule. However, they differ in their choice of (non-trivial) tie-breaking rules. Since PD\(^2\) is the most efficient of the known algorithms and has been the focus of most prior work, we assume its use when deriving the complexity bounds for the scheduler implementation presented later. For this purpose, it is sufficient to know that a PD\(^2\) priority can be determined in constant time and that two PD\(^2\) priorities can be compared in constant time. (See [2] for a detailed explanation of PD\(^2\).)

### 2.2 Extensions

We now summarize several extensions to Pfair scheduling that have been proposed.

**Increasing flexibility.** Srinivasan and Anderson [1, 12] introduced three variants of Pfair scheduling that provide increasing levels of flexibility. In addition, they proved that the PD\(^2\) algorithm correctly schedules any task set satisfying (1) when using each variant. *Early-release-fair* (ERfair) scheduling extends Pfair scheduling by allowing subtasks to be scheduled before the start of their windows. However, each subtask must still use the PD\(^2\) priority associated with its Pfair window placement, *i.e.*, window positions do not change. *Intra-sporadic-fair* (ISfair) scheduling extends ERfair scheduling by allowing subtask releases to be delayed, provided that all future releases are delayed by (at least) the same amount. (Note that this restriction is similar to that applied to job releases under the sporadic task model.) Finally, *generalized-intra-sporadic-fair* (GISfair) scheduling extends ISfair scheduling by permitting subtasks to be absent. However, future subtask releases are not permitted to shift forward in time in response to an absence. Fig. 4 illustrates Pfairness and its variants.

**Dynamic task sets.** In recent work, Srinivasan and Anderson [13] derived conditions under which tasks may leave and join a system. As expected, a task may join as long as (1) continues to hold. Leaving, however, is more complicated and may result in a task’s departure being delayed (for a bounded and predictable duration).

**Supertasking.** In [11], Moir and Ramamurthy proposed the use of *supertasks* to support the static assignment of tasks to processors. In their approach, each supertask replaces a set of *component tasks*, which are bound to a specific processor. Each supertask competes with an ideal weight, *i.e.*, a weight equal to the cumulative weight of its component tasks. Whenever a supertask is scheduled, one of its component tasks is selected to execute according to an internal scheduling algorithm. Unfortunately, Moir and Ramamurthy demonstrated that component-task deadline misses may occur when scheduling supertasks using PF, PD, or PD\(^2\).

In previous work [6, 9], we considered supertasking in the more general context of hierarchal scheduling. In [6], we demonstrated that supertasks can guarantee the timeliness of Pfair component tasks by using a weight that is usually only slightly larger than the ideal weight. We further demonstrated that this weight could be
further reduced by using unfair uniprocessor scheduling algorithms within supertasks, such as the well-known earliest-deadline-first (EDF) algorithm.

**Synchronization.** In other 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 ensures that tasks will never hold locks while preempted. On the other hand, our server-based protocol is designed for critical sections that cannot be efficiently implemented using the timer-based approach. A simple client-server model is used in which a server task replaces a lock and executes all critical sections guarded by that lock. As explained in [7, 8, 9], overheads associated with the synchronization approaches just discussed can be reduced by using supertasking. In essence, this is because the use of a supertask prevents component tasks from executing simultaneously.

### 2.3 Model Specifications

We now formally specify the aligned and staggered models. Let \( t(i, k) \) denote the time 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 \( t(i, k) = i + \frac{k}{M} \). (Recall that only one processor makes scheduling decisions under the aligned model, while the other processors stall.)

### 3 Scheduling under the Staggered Model

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

**Back-to-back scheduling.** Scheduling under the staggered model is actually more restrictive than scheduling under the aligned model. This follows from the fact that quanta from different slots can overlap when slots are staggered, as the A3 and B1 quanta illustrated in Fig. 3(b) (see Sec. 1). Consequently, the scheduler must ensure that tasks scheduled back-to-back (i.e., in consecutive slots) are granted non-overlapping quanta.

Our algorithm satisfies this restriction by guaranteeing that task migrations can occur only after preemptions. Hence, a task that is scheduled back-to-back will execute on the same processor in both slots. This guarantee both ensures the safety of scheduling decisions and allows heavy-weight supertasks (which will often be scheduled back-to-back) to be used to improve cache performance. However, providing such a guarantee is a non-trivial task, as explained below.

**Concept.** To motivate our algorithm’s design, consider the problem of scheduling slot \( t \) on the first processor,
following the execution of task $T$ on that processor in slot $t - 1$. In order to be computationally efficient, the scheduling decision must require only $O(\log N)$ time on each processor. (PD$^2$ schedules all $M$ processors at once in $O(M \log N)$ time under the aligned model [12].) In addition, the decision must respect the back-to-back guarantee described above: if $T$ is one of the tasks selected to execute in slot $t$, then it should continue to execute on the first processor; otherwise, some task that did not execute in slot $t - 1$ should be selected. However, $\Omega(M)$ time is required (in the worst case) just to identify which tasks should execute in slot $t$.

The implication is that the set of tasks scheduled in slot $t$ must be known 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 $t - 1$. Specifically, processor $p$’s scheduler invocation in slot $t$ first selects a task to execute on processor $p$ in slot $t$ from $t$’s scheduling set, and then adds (at most) one task to the scheduling set of slot $t + 1$.

### 3.1 Basic Algorithm

To simplify the presentation of our algorithm, we begin by presenting those procedures needed to support Pfair or ERfair scheduling of static task sets. Later, we present additional procedures to support the remaining Pfair-scheduling extensions. The procedures comprising the basic algorithm are shown in Fig. 5.

**Data structures.** Our algorithm uses five heaps: $SchedNow$, $ReschedNow$, $SchedNext$, $ReschedNext$, and $Eligible$. When scheduling processors in slot $t$, heaps with a $Now$ (respectively, $Next$) suffix store tasks that are selected to execute in slot $t$ (respectively, $t + 1$). In addition, the $Sched$ (respectively, $Resched$) prefix designates that the stored tasks are not (respectively, are) scheduled back-to-back. Hence, the $SchedNow$, $ReschedNow$, $SchedNext$, and $ReschedNext$ heaps are used to efficiently implement the scheduling sets described above. The $Eligible$ heap stores all remaining tasks that can be executed in slot $t + 1$. All heaps are either min- or max-ordered according to task priorities. (The $\preceq$ and $\succeq$ relations, which define the min- and max-orderings, respectively, are defined below.) In addition to the five main heaps, a collection of $Incoming$ heaps store ineligible tasks. To simplify the presentation, we represent this collection using an unbounded array of heaps, where $Incoming[t]$ stores tasks that can be scheduled (next) in or after slot $t$. We assume that each task is initialized and stored in the appropriate $Incoming$ heap during system initialization.

Our algorithm associates a record with each task 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 the task priorities are implemented
using an abstract data type that supports the \( \prec \), \( \preceq \), \( \succeq \), and \( \geq \) operators, where \( \rho_1 \prec \rho_2 \) (respectively, \( \rho_1 \preceq \rho_2 \)) implies that priority \( \rho_1 \) is strictly lower than (respectively, lower than or equal to) priority \( \rho_2 \). We further assume that \texttt{UpdatePriority()} encapsulates the algorithm for updating \texttt{prio} and \texttt{elig}. Abstracting the priority definition in this way allows easy modification.

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

To simplify the presentation, some branches are based on testing 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 simply associating a few additional variables with each task.

**Detailed description.** The \texttt{Initialize} routine is invoked before all other procedures and makes the scheduling decisions for slot 0. First, tasks that become eligible at time 0 are merged into the \texttt{Eligible} heap at line 1. Lines 2–3 then select the tasks that will execute in slot 0.

The \texttt{SelectTask} routine 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.

---

**Figure 5:** Basic staggered scheduling algorithm.
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.

Processor p is scheduled by invoking Schedule(p). Lines 8–11 initialize the scheduling of the next slot by incrementing t, by merging newly eligible tasks into Eligible, and by initializing SchedNow, ReschedNow, SchedNext, and ReschedNext. Lines 12–18 select the task that will execute on the local processor and record the decision in Running. The selected task’s priority is then updated at line 20 and the task is stored in either Eligible or an Incoming heap at lines 21–23, depending on its eligibility. Lines 24–25 then select a task to execute in the slot t + 1. Finally, progress in recorded by updating SchedCount at line 26.

Example. Fig. 6 shows a sample schedule produced by the algorithm in Fig. 5. 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 two additional procedures, shown in Fig. 7, to support tasks leaving and joining the system. Support for such behavior is required when using ISfair or GISfair scheduling, when scheduling dynamic task sets, and when permitting on-line weight changes.

Concerns. Two problems arise from tasks leaving and joining the system. The first problem is that such behaviors require modification of the scheduler’s data structures. Consequently, exclusive access to these structures
must be guaranteed using locks. Since the timing of all scheduler invocations is known a priori, timer-based techniques similar to those discussed in [8] can be applied here as well.

The second problem is that adding and removing tasks can lead to incorrect scheduling decisions. To ensure correct scheduling, we ensure that invariant (I1), shown below, holds after the execution of every procedure.

(11) Upon completion of each procedure invocation, either (i) $|\text{SchedNext}| + |\text{ReschedNext}| = \text{SchedCount}$ and, for all tasks $T \in (\text{SchedNext} \cup \text{ReschedNext})$ and $U \in \text{Eligible}, T.prio \geq U.prio$ holds, or (ii) $|\text{Eligible}| = 0$.

Detailed description. Invoking `Deactivate(T)` within slot $t$ causes task $T$ to be ignored when scheduling slots at and after $t + 1$. In addition, 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 situation 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 $\text{SchedNext}$ or $\text{ReschedNext}$), (ii) $T$ is scheduled in slot $t$ but has not been granted a processor (it is in either $\text{SchedNext}$ or $\text{ReschedNext}$), 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 $\text{SchedNext}$ or $\text{ReschedNext}$ 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 $\text{SchedNext}$ or $\text{ReschedNext}$ at lines 33–36 and then charging $T$ for the unused quanta at line 37. Lines 38–40 handle Case (iii).

Observe that `Deactivate` neither halts executing tasks nor modifies `Running`. The two procedures presented...
here are expected to be used as subroutines when implementing more complex services, such as weight-change requests. Consequently, each procedure takes only those actions that are essential to achieving its goals.

Invoking \texttt{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 \texttt{SchedNext} and \texttt{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 \texttt{Eligible} at line 50.

### 3.3 Time complexity

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

### 4 Impact of Staggering on Prior Work

In this section, we discuss how existing analytical results can be applied to the staggered model.

**Side effects of staggering.** Under staggering, a slot can extend up to \( \Delta = \frac{M-1}{M} \) beyond its endpoint under the aligned model. As a result, slot \( i \) on one processor may overlap slots \( i + 1 \) and \( i - 1 \) on other processors. These characteristics lead to side effects (E1) and (E2), shown below.

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

For instance, a task that receives at least \( c \) units of processor time over \([a, b]\) using the aligned model may only receive \( c - \Delta \) units under staggering. However, \( c \) units are still guaranteed by time \( b + \Delta \).

(E2) Whereas each slot overlaps \( M - 1 \) other slots under the aligned model, each slot overlaps \( 2(M - 1) \) other slots under the staggered model.

**Scheduling theory.** We begin by considering prior work that has focused on independent tasks. Due to independence, tasks are oblivious to the parallel execution of other tasks and, hence, are unaffected by (E2). On the other hand, (E1) impacts two aspects of these results.
Figure 8: In slot \( t \), task \( T \) sends a request to server task \( S \) and then requests suspension. Both events are guaranteed to occur before slot \( t + 1 \) is scheduled (a) under the aligned model, but not (b) under the staggered model.

First, a deadline that is guaranteed under the aligned model may be missed by up to \( \Delta \) under staggering. Hence, the guarantees provided by Pfair scheduling under the staggered model resemble those of using proportional-share scheduling [15] on a uniprocessor. As with that work, we expect such bounded deadline misses to be acceptable in most cases. When this is not the case, deadlines can still be strictly guaranteed, at the expense of some schedulability loss, by increasing each task’s weight so that each strict deadline is shifted one slot ahead of its original position.

Second, events occurring in slot \( t \) may occur after scheduling decisions for slot \( t + 1 \) have been committed, as shown in Fig. 8(b). This can lead to two problems. First, a task requesting suspension in slot \( t \) may have been selected to execute in slot \( t + 1 \) before the request is issued. Second, servers that suspend when their request queues are empty may exhibit poor responsiveness. Fig. 8 illustrates both problems.

**Supertasking.** In [6, 9], the schedulability of component tasks is guaranteed by selecting a supertask weight so that the least amount of processor time guaranteed to a supertask over every possible interval exceeds the total execution-time requirement of all component tasks. Changing from the aligned model to the staggered model only impacts the first of these two quantities, which is independent of (E2).

The impact of (E1) is similar to that described above: a supertask may experience a delay of \( \Delta \) before receiving the processor time guaranteed under the aligned model. As before, this delay may result in bounded (component-task) deadline misses. In this case, strict deadlines can be guaranteed by 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 synchronization 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 retry overhead.\(^2\)

**Timer-based lock synchronization.** Two timer-based locking protocols are presented in [8]: the Rollback Protocol (RP) and the Skip Protocol (SP). Each strives to prevent locks from being obtained when the associated operation is at risk of being preempted before completion.\(^3\) Specifically, a signal is set to occur at a fixed offset within each quantum, as illustrated below. Once the signal occurs, the task’s ability to obtain locks is suspended until it receives its next quantum. Lock requests are stored in a FIFO-ordered queue that uses a protocol-dependent management policy. Under the RP, if a task reaches the end of its quanta before being serviced, then its request will be dropped from the queue. Under the SP, a task’s request is permitted to remain in the queue, but the task will be “skipped over” when granting the lock unless the task is executing, and the task’s ability to obtain locks has not been suspended.

<table>
<thead>
<tr>
<th>Quantum</th>
<th>Freeze Signal</th>
<th>Locking Allowed</th>
<th>Locking Not Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Both protocols guarantee that only executing tasks can hold locks, and hence can block a requesting task \( T \). As with the analysis of lock-free algorithms in [7], the analysis of the SP and the RP presented in [8] assumes that each task \( T \) can be preempted at most once before its critical section completes. Hence, the situation is quite similar to that discussed above: bounds on \( T \)’s worst-case synchronization overhead are computed by considering the interference of \( M - 1 \) tasks under the aligned model and \( 2(M - 1) \) tasks under the staggered model. However, the FIFO-ordering of the queue in this context ensures that no task can block the requesting task twice in the same slot. Hence, each of the \( 2(M - 1) \) interfering tasks must be *unique*.

**Server-based lock synchronization.** As with the timer-based protocols, the request queue is FIFO-ordered under the server-based locking protocol proposed in [8]. Since the analysis of the protocol assumes that every competing task has its request executed before \( T \), the worst-case time required for the server 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.

**The staggering trade-off.** The use of the staggered model represents a trade-off between off-line schedulability and on-line performance. Of the costs mentioned above, those related to task suspensions and synchronization

\(^2\)It is possible to derive tighter bounds by exploiting the fact that only part of each overlapping slot actually overlaps the slot in which \( T \) executes. However, the resulting analysis would be much more complex (and time-consuming) than that presented in [7].

\(^3\)The preemption of lock-holding tasks has the effect of lengthening critical-section durations, often by many orders of magnitude.
will likely be the highest in practice. Hence, the overall cost of staggering will depend mainly on how these two behaviors are handled. We consider each below.

As mentioned in Sec. 1, a task that yields the processor before the next scheduling point is still charged for the wasted processor time. This implies that frequent suspensions will likely lead not only to poor task performance, but also to poor processor utilization. Hence, to achieve reasonable performance under Pfair scheduling, tasks that exhibit such problematic behaviors should not be scheduled at the global level, regardless of the scheduling model. Indeed, this observation was one of the driving motivations behind our work on supertasking [9], which allows problematic tasks to be scheduled as a group rather than individually. The benefit of this approach is that the global scheduler sees 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, than Pfair scheduling. Consequently, both processor utilization and task performance tend to improve when using this approach. Given these observations, task suspensions are expected to occur infrequently, if at all, in the global schedule and, hence, should contribute little to the cost of staggering.

By similar reasoning, task 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 under the staggered model should be quite reasonable in practice.

Based on the above considerations, use of the staggered model is expected to result in only slight schedulability loss in practice. Furthermore, simulation results presented in the next section suggest that substantial performance gains can be achieved through staggering. Hence, we expect staggering to be an effective technique for improving system performance without sacrificing predictability.

5 Experimental Results

In this section, we present the results of two empirical studies conducted to determine the cost of bus contention and the efficiency of the scheduling algorithm presented in Sec. 3.

5.1 Characterizing the Cost of Bus Contention

In this subsection, we characterize the cost of scheduling all processors simultaneously and the performance improvement provided by staggering.

Experimental setup. All experiments described in this subsection use the Limes parallel-processing simulator [10]. Limes is intended to simulate the result of a parallel-processing application executing close to the
hardware level. As such, it does not provide any thread preemption or scheduling services, but permits the hardware-level aspects of the system to be more easily controlled than the more complex simulators that we considered. Due to the limitations of Limes, our simulations consisted of simulating the effect of preemption and later resuming a task rather than actually taking these actions. In particular, each cache was initially filled with dirty cache lines. Each task was then assigned multiple arrays, each of which matched the cache in size and used the same cell-to-line mapping. Each task’s execution consisted of incrementing cells in one array while occasionally polling the simulation time. Whenever a slot change was detected, the polling task changed the array being used. In this way, periodic working-set changes were used to simulate the effect of preemption.

Each experiment was conducted for values of $M$ (the processor count) in the range 1, . . . , 16 and reported the behavior of the system across three slots. The simulated environment consisted of 200 MHz i86 processors using a 10 millisecond tick size and a simple blocking, direct-mapped cache with 256 KB capacity and 32-byte lines. The cache was configured to use the MESI coherency protocol. In designing our experiments, our goal was to measure only bus contention caused by task preemptions. Hence, we tried to minimize the impact of the coherency protocol by avoiding both actual and false data sharing.

Bus contention was measured by counting the number of pending bus requests at each simulation cycle. Since only one request could be serviced in each simulated cycle, the presence of $k$ pending requests implied that $k - 1$ tasks wasted that cycle waiting for bus access. (Note that we did not count the cycles required to service a task’s bus requests as wasted cycles.)

Relevance. In the simulations considered here, it is assumed that tasks process large data sets, where “large” is measured relative to the cache size. This assumption is motivated by the fact that many real-time multiprocessor applications consist almost exclusively of such tasks. For instance, both signal-processing and virtual-reality applications are based on the processing and manipulation of large data sets. Hence, we believe that the scenarios considered here are reflective of situations that can arise in real multiprocessor systems.

Worst-case contention under the aligned model. The first experiment estimated the worst-case bus contention possible under the aligned model\(^4\) when using data sets that can be completely cached. When using aligned slots, the worst case occurs when each task’s reaction to being scheduled is to write to each cache line in its working set immediately. Fig. 9(a) shows the average number of cycles lost per task in this scenario. (The bus traces for the $M = 8$ case were shown earlier in Sec. 1.) To place the performance loss in perspective, the interval over which the statistics were gathered consisted of 6,000,000 cycles. Hence, a loss of 3,000,000 cycles

\(^4\)This experiment does not characterize the worst case for staggered scheduling.
Figure 9: (a) Cycles lost to bus contention when each task’s working set completely fills the cache and each cache line is systematically written. (b)–(c) Cycles lost to bus contention as each task’s working-set size is varied and each task randomly writes to locations in its working set.

implies that, on average, each task spent 50% of its execution time waiting for access to the shared bus.

Both curves in Fig. 9(a) illustrate three performance trends. First, when the processor count is low (two), the delays between consecutive bus requests on each processor are sufficiently large to prevent significant contention for the bus. However, as the processor count increases, these delays can no longer compensate for the increasing aggregate rate of requests, and the average waiting time increases linearly as processors are added. Note that the increase is far steeper under the aligned model, as expected. As the processor count exceeds ten, the request rate becomes so high under both models that the bus cannot service all requests, i.e., the bus becomes overloaded. Consequently, the addition of more processors has the effective of throttling down each task’s request rate as the total number of requests serviced per unit time remains constant.
Random-access contention. Our second experiment gauged how performance scales under each model when access patterns are random and the size of the working sets vary. In this experiment, each task randomly selected which cell values to increment and were only permitted to select cells in a fraction $\alpha$ of the array. Selecting cells randomly results in an initial burst of bus activity at the start of each slot; the request rate of each task then drops off gradually over time as the probability of referencing an uncached line decreases. The experiment was repeated for each value of $\alpha$ in $\{0.2k \mid 1 \leq k \leq 5\}$.

The results of these experiments are shown in Fig. 9(b)–(c). As shown, staggering results in significantly less loss for all values of $\alpha$. This experiment was repeated several times with different random number seeds and produced graphs that were virtually identical in each run. (Although confidence intervals would be preferable here, the time required to run a single experiment makes this impractical.)

5.2 Evaluating Staggered Scheduling

Our second series of experiments compared the per-slot scheduling overhead of the staggered algorithm presented in Sec. 3 to that of the master/slave algorithm proposed in [1]. To make the comparison as fair as possible, the master/slave algorithm was modified to provide the back-to-back scheduling guarantee described in Sec. 3. The purpose of this comparison was to determine whether the design of the staggered algorithm is practical, and whether the staggered algorithm will actually produce less overhead in practice.

Experimental setup. Two scenarios were tested: fully cached uniprocessor execution and realistically cached multiprocessor execution. The goal of the first scenario was to estimate the relative overhead of these algorithms under worst-case analysis. Recall that the per-slot overhead under staggered (respectively, master/slave) scheduling is the time required to execute Schedule once (respectively, to schedule all processors). Because a scheduling algorithm’s memory-reference patterns will be difficult to predict accurately, worst-case analysis typically relies on pessimistic estimates of the cache behavior. In the extreme case, every reference to a heap element will be assumed to miss in the cache, resulting in an effectively uniform memory-access cost. Unfortunately, we were unable to secure a cacheless machine to measure performance under this assumption. Instead, we structured the experiment so that virtually all memory references would hit in cache. Though this results in much lower memory-access costs, the costs are approximately uniform. Hence, the relative performance observed in this setting should provide a reasonable estimate of the relative performance in cacheless systems.

The goal of the second scenario was to estimate average-case relative overhead under realistic caching conditions. In practice, master/slave scheduling will make better use of the cache on average because all scheduling

5Platforms: 700-MHz Dell PC running Red Hat Linux 2.4, SGI Reality Monster (32 300-MHz R10000s) running Irix 6.5.
decisions are made on one processor. Indeed, master/slave scheduling can experience a cold cache only at the start of each scheduler invocation, while staggered scheduling may experience a cold cache each time Schedule is invoked. The effect of a cold cache was experimentally simulated by migrating the scheduler among a set of $M$ processors in a round-robin fashion. Specifically, the master/slave (respectively, staggered) scheduler was migrated after each scheduler (respectively, Schedule) invocation.

For each scenario, 1,000 sets of independent tasks were randomly generated for each pairing of $N$ and $M$, where $N \in \{10n \mid 1 \leq n \leq 50\}$ and $M \in \{2, 4, 8, 16\}$, and the average execution time of each scheduler was measured. From these measurements, the ratio of the average per-slot overhead of the master/slave algorithm to that of the staggered algorithm was computed. Although 99% confidence intervals were computed for each sample point, they were too small to be visible in the graphs and hence have been omitted.

**Results.** Fig. 10(a) shows the result of the uniprocessor experiment. As shown, the staggered algorithm produces an improvement that is close to, and often matches, the factor-of-$M$ improvement suggested by Fig. 3. Hence, these results suggest that the staggered algorithm is efficient and should result in significantly lower per-slot overhead under worst-case analysis.

Fig. 10(b) shows the result of the multiprocessor experiment. As shown, caching effects close the performance gap between the two algorithms substantially. Despite this, staggered scheduling still improves upon master/slave scheduling in all cases. Hence, these experiments (combined) suggest that the staggered algorithm should improve upon the master/slave algorithm under both average-case and worst-case analysis.

---

To reduce the simulation overhead, the scheduler thread was not actually migrated. Instead, $M$ scheduler threads were created and synchronized with simple spin locks so that only one was permitted to execute at each time instant.
6 Related Work

Because Pfair scheduling is a relatively recent development, most prior work relating to this approach has focused on extending its theoretical limits rather than producing an efficient implementation. Indeed, the only prior work that has considered the latter problem (of which we are aware) is that of Chandra, Adler, and Shenoy [5]. That work focused on improving the average-case performance of Pfair scheduling in a general-purpose operating system. Consequently, the scheduling algorithm proposed in [5] was only evaluated experimentally and hence does not guarantee predictability. In contrast, the goal of our work is to produce an implementation that is both efficient and capable of providing real-time guarantees. Hence, we have only considered performance optimizations that do not sacrifice worst-case predictability.

7 Conclusion

In this paper, we considered the problem of supporting Pfair scheduling on a bus-based SMP architecture. Although the architecture complements Pfair scheduling in many ways, simulations presented herein suggest that preemption-related bus contention can retard the progress of tasks significantly. To alleviate this problem, we proposed a staggered scheduling model for Pfair scheduling in which scheduling points are uniformly distributed over time. Under the hypothesis that preemption-related bus traffic occurs primarily at the start of each quantum, the proposed model should provide a considerable improvement over the current model.

To facilitate the use of the proposed model, we presented an efficient scheduling algorithm that both supports the new model and introduces less scheduling overhead than previously proposed algorithms. In addition, the presented algorithm provides a back-to-back scheduling guarantee that allows cache performance to be improved through the use of supertasks. Finally, we explained how to adapt existing results to the proposed model, thereby demonstrating the predictability of scheduling under the new model and characterizing its cost.

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


