Task Reweighting on Multiprocessors: Efficiency versus Accuracy[*]

Aaron Block and James H. Anderson

Department of Computer Science

University of North Carolina at Chapel Hill

Abstract

We consider the problem of *task reweighting* in fair-scheduled multiprocessor systems wherein each task's processor share is specified using a *weight*. When a task is reweighted, a new weight is computed for it that is then used in future scheduling. Task reweighting can be used as a means for consuming (or making available) spare processing capacity. The responsiveness of a reweighting scheme can be assessed by comparing its allocations to those of an ideal scheduler that can reweight tasks instantaneously. A reweighting scheme is *fine-grained* if any additional per-task "error" (in comparison to an ideal allocation) caused by a reweighting event is constant. Similarly, a reweighting scheme is *coarse-grained* if the additional "error" per reweighting event is non-constant. While fine-grained reweighting produces only a small amount of error when changing task weights, it has a worst-case time complexity of $\Theta(N log N)$, where N is the number of tasks. If the number of tasks exceeds the number of processors, then such time complexity is larger than that of coarse-grained reweighting, which is $\Theta(M log N)$, where M is the number of processors. In this paper, we construct two new reweighting algorithms that are hybrids of fine- and coarse-grained reweighting that have time complexity less than $\Theta(N log N)$ and produce less error than current coarse-grained techniques. We also present an experimental evaluation of these scheme to compare their relative advantages.

Keywords: Real-time, Fair-scheduling, Adaptive, Multiprocessor, Pfair

[*]Work supported by NSF grants CCR 9972211, CCR 9988327, ITR 0082866, and CCR 0204312. The first author was also supported by an NSF fellowship.

1 Introduction

Two trends are evident in recent work on real-time systems. First, *multiprocessor* designs are becoming increasingly common. This is due both to the advent of reasonably-priced multiprocessor platforms and to the prevalence of computationally-intensive applications with real-time requirements that have pushed beyond the capabilities of single-processor systems. Second, many applications now exist that require the ability to *adaptively* react to external events within short time scales by adjusting task parameters, particularly *processor shares.* Examples of such applications include systems that track people and machines, many computer-vision systems, and signal-processing applications such as synthetic aperture imaging.

One such adaptive application is the Whisper tracking system designed at the University of North Carolina to perform full-body tracking in virtual environments [17]. As depicted in Fig. 1, Whisper tracks users by attaching speakers that emit white noise to various body points (hands, feet, *etc.*). Microphones located on the ceiling receive these signals and a tracking computer calculates each speaker's distance from each microphone by measuring the associated signal delay. The correlation computations required to do this are computationally expensive. Indeed, tracking more than a few speakers *requires* a multiprocessor system. Like many tracking systems, Whisper uses *predictive techniques* to track objects. When tracking an object (speaker), the computational cost of making the "next" prediction depends on the accuracy of the previous one, as an inaccurate prediction requires a larger space to be searched. Thus, the processor shares of the tasks that are deployed to implement these tracking functions vary with time. In fact, the variance can be as much as *two orders of magnitude* and can change within *time scales as short as 10 ms.* When changing the shares of tasks, there is a natural tension between accuracy and time-complexity overhead. The development of multiprocessor techniques for enacting such share adaptations that balance these two concerns is the subject of this paper.

The most common approach for scheduling tasks in multiprocessor systems is to *partition* the tasks among the processors. The tasks on each processor can then be scheduled using *uniprocessor* scheduling algorithms, as depicted in Fig. 2(a). Partitioning eliminates overheads due to task migrations. However, the problem of assigning tasks to processors is equivalent to bin packing, which is NP-hard in the strong sense. Also, partitioning

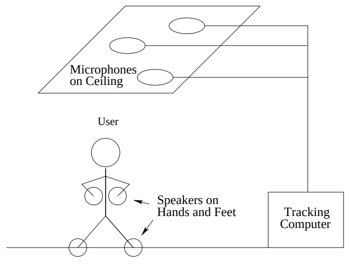

Figure 1: The Whisper tracking system.

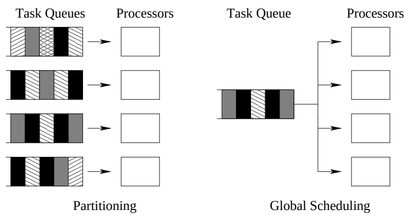

Figure 2: **(a)** Partitioning. **(b)** Global scheduling.

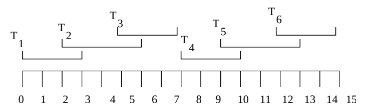

Figure 3: The first six Pfair windows for a task of weight 3/8.

is inherently non-optimal. In an adaptive system like Whisper, these problems are even more pronounced. This is because adjustments to task shares may necessitate *frequent* re-partitionings. Even if acceptable heuristics can be found for this purpose, re-partitioning forces tasks to migrate, which is the very thing partitioning approaches are designed to avoid.

Because task migrations are difficult to preclude when supporting share adaptations, we consider *global* scheduling algorithms that allow migrations, as depicted in Fig. 2(b). We also assume that the platform under consideration is a tightly-coupled shared-memory system. In such a system, the "cost" of a migration is simply a loss of cache affinity. We further constrain the discussion by considering a particular class of global scheduling algorithms known as *fair* scheduling algorithms. Under fair scheduling, correctness is defined by comparing to an *ideal* scheduler that can guarantee each task *precisely* its required share over any time interval. Such an ideal scheduler can instantaneously enact share changes, but is impractical to implement, as it requires the ability to preempt and swap tasks at arbitrarily small time scales. Practical schemes are designed so that share allocations track the ideal with only bounded "error." In a fair-scheduled system, we measure the efficacy of an adaptive policy by the additional per-task "error" (in comparison to the ideal allocation) caused by a task share change; we use the term *drift* when referring to this source of error. We have chosen to focus on fair-scheduled systems for two reasons: **(i)** fair scheduling algorithms (in particular, Pfair algorithms, as discussed below) are the only known way to optimally schedule recurrent (*i.e.*, periodic, sporadic, or rate-based) real-time tasks systems on multiprocessors [3, 6, 7, 14]; **(ii)** prior work on *uniprocessor* notions of fairness has resulted in a number of related schemes that can be used to change task shares with small drift and low time complexity [8, 9, 10, 13]. We seek to show that similar adaptive policies exist on multiprocessors.

The various notions of fairness considered in this paper are derivatives of the Pfairness constraint of Baruah *et al.* [6]. Under Pfair scheduling, each task executes at a uniform rate, while respecting a fixed allocation quantum. Each task's rate is specified by a rational *weight* in the range (0,1], which also defines its desired processor share. As explained in detail later, uniform rates are ensured by requiring the allocation error for each task to be always less than one quantum. Due to this requirement, each task T is effectively subdivided into quantum-length *subtasks* $T_1, T_2, \ldots$ that must execute within *windows* of approximately equal lengths: if some T_i executes outside of its window, then T's error bounds are exceeded. Fig. 3 shows an example window layout. Pfair scheduling algorithms schedule tasks by their deadlines, where a subtask's "deadline" is given by the end of its window. $\Theta(M\,log N)$ time is required to schedule M processors, where N is the total number of tasks. (Throughout the paper, we assume $N > M$, because otherwise each task can be trivially scheduled on a dedicated processor.)

Under Pfair scheduling, a task's share is changed through a process called *reweighting.* Reweighting schemes change the weight of a task and, as a result, change its future subtask releases and deadlines. We define the *time*

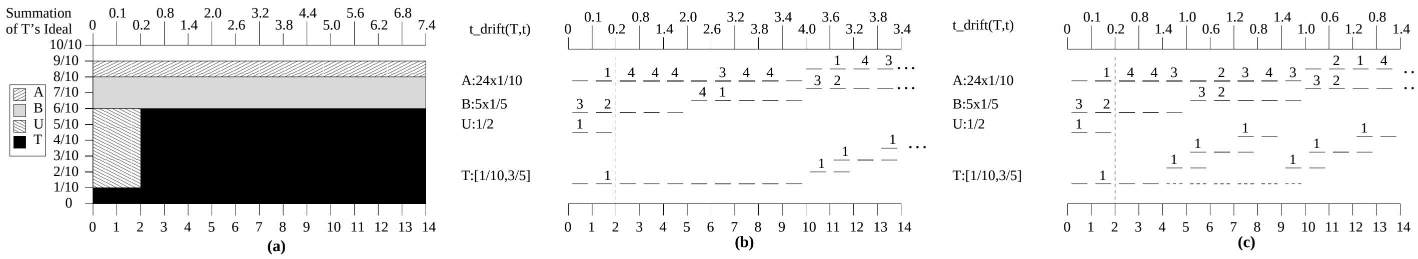

Figure 4: A four-processor system with a set A of 24 tasks of weight 1/10, a set B of 5 tasks of weight 1/5, a task U of weight 1/2 that leaves at time 2, and a task T of weight 1/10 that should increase to 3/5 at time 2. The numbers in each time slot denote the number of tasks that are scheduled in each slot. The insets show T's allocation in **(a)** the ideal system, **(b)** under Pfair scheduling with coarse-grained reweighting, and **(c)** under Pfair scheduling with fine-grained reweighting. In (a), T's cumulative allocation is shown across the top. In (b) and (c), the difference between T's actual and ideal allocation is shown across the top (*i.e.*, the total drift for task T up to time t).

complexity of such a scheme to be the worst-case time required to reweight all N tasks on M processors. Such a scenario can occur in Whisper if a user's movements suddenly become erratic thus causing tracking predictions to become inaccurate. Two kinds of multiprocessor reweighting schemes have been proposed in prior work: *coarse-grained* and *fine-grained* reweighting. Coarse-grained reweighting schemes can have an arbitrarily large amount of drift per reweighting event, but can be implemented in $\Theta(M\,logN)$ time. Fine-grained reweighting schemes incur only a constant amount of drift per reweighting event, but have a time complexity of $\Theta(N\,logN)$. The goal of this paper is to construct a hybrid scheme that combines the benefits of both types of reweighting schemes. Before proceeding, we present a brief overview of coarse- and fine-grained reweighting techniques.

Coarse-grained reweighting. Srinivasan and Anderson [15] have given sufficient conditions (described in detail in Sec. 2) under which tasks may dynamically join and leave a running Pfair-scheduled system without causing any missed deadlines. A task may join such a system if doing so does not cause the system to be over-utilized. Leaving, however, is more complicated. In particular, if a leaving task is over-allocated (in comparison to its ideal allocation), then it is necessary to *delay* its actual departure. Without such delays, a task could artificially boost its share by repeatedly leaving and joining, causing missed deadlines.

A *coarse-grained* reweighting policy is simple to obtain from these rules: a task T changes its weight from v to u by leaving with weight v and rejoining with weight u. This policy fails to be fine-grained because of potential leaving delays. To see why, consider the four-processor schedule in Fig. 4(b). The depicted system consists of a set A of 24 tasks of weight 1/10, a set B of five tasks of weight 1/5, a task U of weight 1/2 that leaves at time 2, and a task T of weight 1/10 that should increase to 3/5 at time 2 (due to the capacity freed by U's leaving). Because of the restrictions placed on leaving (as specified in Sec. 2), T cannot leave before time 10, the end of its first window. As a consequence, T's actual allocation drifts from its ideal by four quanta over the interval $[2, 10]$. This example can be generalized to show that *arbitrarily large* drift is possible with only one weight change.

By delaying a task's reweighting event until the next time it is scheduled, reweighting using leave/join rules can be implemented in $\Theta(M\,logN)$ time. Since only M tasks are scheduled in each time slot, this method guarantees that only M tasks can be reweighted at any time. Compared to instantly enacting weight changes, delaying reweighting events may increase the amount of drift incurred per reweighting event. However, with or without such delays, the maximal drift incurred can be arbitrary large.

Fine-grained reweighting. Recently, Block and Anderson [1] presented sufficient conditions (described in detail in Sec. 3) under which tasks may dynamically change their weights in a running Pfair-scheduled system while incurring only a constant amount of drift per reweighting event. These rules (unlike the leave/join rules) reweight a task by immediately changing the release time of its next-released subtask and/or the deadline of its current subtask. This instantaneous behavior guarantees that the amount of drift incurred per reweighting event is constant. As an illustration, consider Fig. 4(c), which depicts a schedule involving the same system considered earlier, except that now task T increases its weight via the fine-grained rules. Because the weight change at time 2 immediately affects the system, the amount of drift incurred is at most 1.4. Since these rules need to be immediately enacted, we *cannot* use the same "delaying technique" used in coarse-grained reweighting to reduce time complexity. Indeed, as we discuss in Sec. 3, the time complexity of *any* Pfair-scheduled fine-grained reweighting scheme is $\Omega(N\,logN)$.

New approach: Hybrid schemes. There are many scenarios under which the error bounds provided by fine-grained reweighting are not necessary and yet those provided by coarse-grained reweighting are too weak. In this paper, we present two new reweighting schemes that have (non-constant) bounded drift and have a time complexity lower than $\Theta(N\,logN)$ time. These schemes are introduced in the following way. After first presenting a more careful review of prior work in Sec. 2, we present our new schemes in Sec. 3, and establish both drift and time-complexity bounds for each. In Sec. 4, we assess the efficacy of these schemes by presenting results from an experimental evaluation.

2 Preliminaries

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

Under Pfair scheduling, processor time is allocated in discrete time units, called *quanta*; the time interval $[t, t+1)$, where t is a nonnegative integer, is called *slot* t. (Hence, time t refers to the beginning of slot t.) The sequence of allocation decisions over time defines a *schedule*. Formally, a schedule S is a mapping $S : \tau \times \mathcal{N} \mapsto \{0, 1\}$, where τ is a set of tasks and $\mathcal{N}$ is the set of nonnegative integers; $S(T, t) = 1$ iff T is scheduled in slot t.

The notion of a Pfair schedule is defined by comparing to an ideal schedule that allocates $wt(T)$ processor time to task T in each slot. Deviance from the ideal schedule is captured by the concept of *lag*. The *lag of task T at time t*, $lag(T, t)$, is defined as $wt(T) \cdot t - \sum_{u=0}^{t-1} S(T, u)$. A schedule is *Pfair* iff $(\forall T, t :: -1 < lag(T, t) < 1)$. Informally, each task's allocation error must always be less than one quantum.

Each quantum of a task's execution, henceforth called a *subtask*, must be allocated without violating the lag bounds above. We denote the i^{th} subtask (*i.e.*, i^{th} quantum of allocation) of task T as T_i, where $i \geq 1$. Associated with subtask T_i is a *pseudo-release* $r(T_i)$ and *pseudo-deadline* $d(T_i)$ defined as follows.

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

(For brevity, we often drop the prefix "pseudo-.") It can be shown that if each subtask T_i is scheduled in the interval $w(T_i) = [r(T_i), d(T_i))$, termed its *window*, then $(\forall T, t :: -1 < lag(T,\ t) < 1)$. For example, in Fig. 6(a), $r(T_2) = 3$, $d(T_2) = 7$, and $w(T_2) = [3, 7)$. (This figure also depicts per-slot "flow values," which are considered below.) Thus, T_2 must be scheduled in slot 3, 4, 5 or 6. (Tasks must execute sequentially, so if T_1 is scheduled in slot 3, then T_2 must be scheduled in slot 4, 5, or 6.)

IS model. The intra-sporadic (IS) task model [14] generalizes the well-known sporadic task model [12] by allowing subtasks to be released late. This extra flexibility is useful in many applications where processing steps may be delayed. Fig. 6(b) illustrates the Pfair windows of an IS task. We consider a task T to be *active* at time t if there exists a subtask of T_k such that $r(T_k) \leq t < d(T_k)$, and passive otherwise. For example, in Fig. 6(b), T is passive in slot 4, and active for every other time slot. Each subtask T_i of an IS task has an *offset* $\theta(T_i)$ that gives the amount by which its release has been delayed. By (1), $r(T_i) = \theta(T_i) + \left\lfloor \frac{i-1}{wt(T)} \right\rfloor$ and $d(T_i) = \theta(T_i) + \left\lceil \frac{i}{wt(T)} \right\rceil$.

Offsets are constrained so that the separation between any pair of subtask releases by a task is at least the separation between those releases if the task were periodic. Formally, the offsets satisfy the following property: $k \geq i \Rightarrow \theta(T_k) \geq \theta(T_i)$.

Scheduling algorithms. Three Pfair scheduling algorithms are optimal for scheduling IS tasks on an arbitrary number of processors: PF [6], PD [7], and PD^2 [3]. Each prioritizes subtasks on an earliest-pseudo-deadline-first (EPDF) basis, but they use different tie-breaks. For the case wherein all task weights are at most 1/2 (our focus here), the three are the same and use just single tie-break, $b(T_i)$, which is defined as $\lceil i/wt(T) \rceil - \lfloor i/wt(T) \rfloor$. In a periodic task system, $b(T_i)$ is 0 if T_i's window does not overlap T_{i+1}'s, and is 1 otherwise. For example, in Fig. **??**, $b(T_i) = 1$ for $1 \leq i \leq 4$ and $b(T_5) = 0$. If two subtasks have equal deadlines, then a subtask with a b-bit of 1 is favored over one with a b-bit of 0.

The group deadline is needed in systems containing tasks with window length of two (*i.e.*, heavy tasks). A

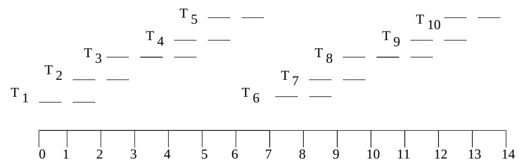

Figure 5: The first few windows of a heavy periodic task of weight 5/7.

periodic task satisfying this condition has group deadlines at the end of each slot that is contained only within a single window. For example, the task in Fig. 5 has group deadlines at times 4, 7, 11, and 14. The group deadline of a subtask T_i, denoted $D(T_i)$, is the earliest time t, where $t \geq d(T_i)$, such that t is a group deadline of T. For example, in Fig. 5 $D(T_2) = 4$ and $D(T_3) = 7$. For an IS task, the group deadline is defined similarly assuming that all future subtasks are released as early as possible. If T is a light task, then $D(T_i)$ is defined to be zero. If two subtasks have equal deadlines and b-bits, then the task with the larger group deadline is favored. All other ties are broken arbitrarily. (See [5] for a more detailed explanation.)

Lag and flow. The lag of an IS task can be defined in much the same way as for periodic tasks [14]. Let *ideal*$(T,\,t)$ denote the share that T receives in a fluid schedule in $[0,t)$. Then,

$$lag(T,\,t) = ideal(T,\,t) - \sum_{u=0}^{t-1} S(T,u). \tag{2}$$

Before defining *ideal*$(T,\,t)$, we define *flow*$(T,\,u)$, which is the share assigned to task T in slot u.[1] *flow*$(T,\,u)$ is defined in terms of a function f that indicates the share assigned to each *subtask* in each slot. f can be defined using an arithmetic expression, but we have opted instead for a more intuitive pseudo-code-based definition in Fig. 7. Observe that, while subtask T_i is active, $f(T_i,t)$ usually equals $wt(T)$, but may be less than $wt(T)$ at $r(T_i)$ and $d(T_i) - 1$ in order to guarantee, respectively, that T's cumulative per-slot share is not greater than $wt(T)$ and that T_i's share across all slots is one. Some example f values are given in Fig. 6. The following two properties follow from f's definition.

F1: For all time slots t, $\sum_{T_i \in T} f(T_i,t) \leq wt(T)$.

F2: For any subtask $T_i \in T$, $\sum_t f(T_i,t) = 1$.

For example, in Fig. 6(a), $\sum_{T_i \in T} f(T_i,3) = \frac{1}{16} + \frac{4}{16}$, and $\sum_t f(T_2,t) = \frac{4}{16} + \frac{5}{16} + \frac{5}{16} + \frac{1}{16} = 1$.

flow$(T,\,t)$ is defined as $\sum_i f(T_i,t)$. For example, in Fig. 6(a), *flow*$(T,\,3) = f(T_1,3) + f(T_2,3) + f(T_3,3) + ... = \frac{1}{16} + \frac{4}{15} + 0 + 0 + ... = \frac{5}{16}$.

[1]In [4], a flow-graph construction is used to show that an IS task system τ is feasible on M processors iff $\sum_{T \in \tau} wt(T) \leq M$. The term "flow" arises because of connections to that proof.

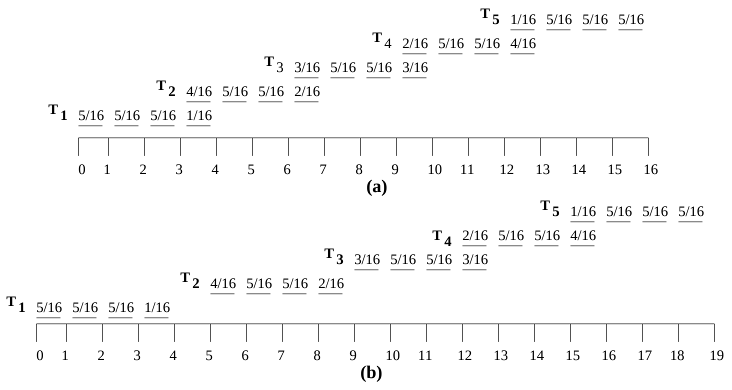

Figure 6: Per-slot f values for a **(a)** periodic and **(b)** IS task of weight 5/16.

```
$f(T_i$: subtask, $t$: integer)
1:  if $t < r(T_i) \vee t > (d(T_i) - 1)$ then
2:      $f(T_i, t) := 0$
3:  else if $t = r(T_i)$ then
4:      if $i = 1 \vee b(T_{i-1}) = 0$ then
5:          $f(T_i, t) := wt(T)$
6:      else
7:          $f(T_i, t) := wt(T) - f(T_{i-1}, d(T_i) - 1)$
8:      fi
9:  else if $t = d(T_i) - 1$ then
10:     $f(T_i, t) := max(1 - \sum_{q=0}^{t-1} f(T_i, q), wt(T))$
11: else
12:     $f(T_i, t) := wt(T)$
13: fi
```

Figure 7: Pseudo-code defining $f(T_i, t)$.

$ideal(T, t)$ is defined as $\sum_{u=0}^{t-1} flow(T, u)$. Hence, by (2), $lag(T, t+1) = \sum_{u=0}^{t}(flow(T, u) - S(T, u)) = lag(T, t) + flow(T, t) - S(T, t)$.

Dynamic task systems. The leave/join conditions of Srinivasan and Anderson [15] mentioned earlier, and a theorem concerning them, are stated below. (Recall that M denotes the number of processors.)

J: (*joining*) A task T can join at time t iff the sum of the weights of all tasks after joining is at most M.

L: (*leaving*) Let T_i denote the last-scheduled subtask of T. Then, T can leave at time t iff $t \geq d(T_i) + b(T_i)$ holds. If T is heavy, then T can leave at time t iff $t \geq D(T_i)$.

Theorem 1 ([15]). PD^2 *correctly schedules any feasible dynamic IS task system satisfying J and L.*

3 Reweighting Algorithms

We consider reweighting algorithms wherein a task T's weight at time t is required to satisfy the following property:

P: $(\forall T, t :: 0 < minwt(T) \leq wt(T, t) < maxwt(T) \leq 1)$,

where $minwt(T)$ and $maxwt(T)$ denote, respectively, the minimum and maximum allowable weight of task T. As a shorthand, we use the notation $T{:}[x, y]$ to denote a task T with $minwt(T) = x$ and $maxwt(T) = y$, and $T{:}z$ to denote a task T with $minwt(T) = maxwt(T) = z$.

The accuracy of a reweighting algorithm $\mathcal{A}$ is measured by the amount of "total drift" that is incurred [1]. We define *total drift* as

$$t\_drift_{\mathcal{A}}(T, t) = \int_0^t wt_0(T, u)du - allocation_{\mathcal{A}}(T, t),$$

where $allocation_{\mathcal{A}}(T, u)$ is the total allocation under algorithm $\mathcal{A}$ for task T up to time t, and $wt_0(T, u)$ is equal to $wt(T, t)$ if T is active at t and 0 if T is inactive at t. The relationship between $\int_0^t wt_0(T, u)du$ and $t\_drift_{\mathcal{A}}(T, t)$

Name	Time Complexity	Maximal Drift per Change	"Top k" Drift per Change
Fine-Grained	$\Theta(N\,log\;N)$	2	2
k-Fine-Grained	$\Theta((M+k)log\;N)$	$min\left(\frac{2\cdot Q(T)}{minwt(T)}, \frac{Q(T)\cdot N}{k}+2\right)$	2
Lazy	$\Theta(M\,log\;N)$	$\frac{2\cdot Q(T)}{minwt(T)}$	$\frac{2\cdot Q(T)}{minwt(T)}$
Leave/Join	$\Theta(M\,log\;N)$	$\frac{2\cdot Q(T)}{minwt(T)}$	$\frac{2Q(T)}{minwt(T)}$

Figure 8: Comparison of reweighting schemes. The function $Q(T) = maxwt(T) - minwt(T)$.

is illustrated in Fig. 4, considered earlier.

Time-complexity lower bound. Xu and Lipton [18] have shown that, for a uniprocessor system, the time complexity of any reweighting policy that allows a task's to weight to change to any arbitrary value with only constant error is $\Omega(N logN)$, where N is the number of tasks.[2] Their proof is based on a reduction to sorting. Consequently, the sorting lower bound, $\Omega(N logN)$, applies to reweighting algorithms as well. Therefore, for the remainder of this section, we discuss how best to tradeoff the accuracy of an allocation policy with the time complexity required. To this end, we propose two new reweighting algorithms: *lazy reweighting* and *k-fine-grained reweighting*. Lazy reweighting is a coarse-grained reweighting algorithm that applies Block and Anderson's reweighting rules to a task only when it is scheduled in ordered to produce better results than leave/join reweighting. k-fine-grained reweighting is a hybrid of lazy reweighting and fine-grained reweighting, where in each time slot, k tasks in addition to those scheduled in that slot are reweighted. A comparison of leave/join, lazy, k-fine-grained, and fine-grained reweighting is presented in Fig. 16.

Implementing PD^2. Before discussing these various reweighting schemes, we review an implementation [7] of PD^2 that can make the scheduling decisions for each time slot with a time complexity of $\Theta(M logN)$. In this implementation, we maintain one priority queue for each time slot at which future subtask releases are to occur. Each priority queue contains those subtasks that are to be released in its associated time slot; these queues are called *release heaps*. We also maintain one additional priority queue called the *master heap*, which contains the set of subtasks that are ready to be scheduled. All priority queues are implemented as binomial heaps and sorted in accordance with the PD^2 priority rules.

Using these heaps, PD^2 is implemented by the following steps. Before time slot 0, the first subtask of each task is inserted into either the master heap or a release heap, depending on whether it commences execution at time 0. (Note that this step takes $\Theta(N logN)$ time, but since it occurs prior to any scheduling decisions, it is not

[2]Technically, this lower bound has been established only for the *decision tree model with comparison tests*, wherein a program is represented as a tree where each node is labeled with a predicate of a list of inputs to the tree, and the only allowable predicates are comparisons between the elements in the list of inputs.

include in the *scheduling* time complexity.) The following steps are then performed in each time slot t ($t \geq 0$).

1. The M highest-priority subtasks from the master heap are removed (if that many are eligible).
2. For each subtask T_i just removed, the deadline and release of its successor T_{i+1} are calculated.
3. If $r(T_{i+1}) > t + 1$, then T_{i+1} is inserted into the release heap for time $r(T_{i+1})$, otherwise T_{i+1} is inserted into the master heap.
4. The next time slot's release heap is merged with the master heap.

An example of the implementation of PD^2 for time steps 0 to 3 is given in Fig. 9 for a one-processor system with three tasks, U with weight 3/11, V with weight 2/7, and W with weight 1/6. Notice that at time 3, the release heap for time 3 is merged with the master heap.

Fine-grained reweighting. Fine-grained reweighting (unlike coarse-grained reweighting) changes the weight of a task, by changing its *next* subtask deadline and/or its next subtask release time. Block and Anderson [1] have given a set of three rules under which a task may change its weight while guaranteeing constant drift per weight change. (Due to the technical nature of these rules, we offer an intuitive explanation and refer the reader to Block and Anderson's original paper for details [1].) There are two factors in choosing which of the three rules to use to change a task's weight: **(i)** whether the task's current weight is greater than 1/2; and **(ii)** whether the currently-active subtask of a task has been scheduled prior to its weight change. We refer the reader to the appendix for a discussion on tasks that have a current weight greater than 1/2.

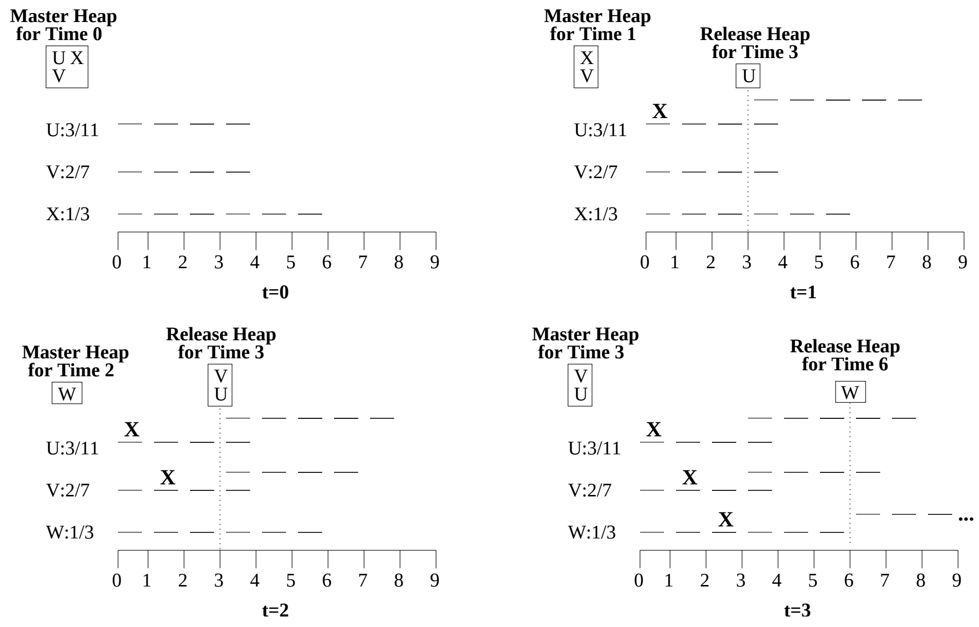

Figure 9: An example of the implementation of PD^2 on a one-processor system with three tasks: U with weight 3/11, V with weight 2/7, and W with weight 1/6. Each X represents a time slot in which a task is scheduled. Each box above the schedule denotes either a release or master heap. The entries in each box denote the tasks with subtasks in the corresponding heap, in sorted order.

Let T_i be the currently-active subtask of a task T that is changing its weight at time t_c from u to v. We say that T is *flow-changeable at time t_c from weight u to v* if T_i is scheduled before t_c, and otherwise is *omission-changeable at time t_c from weight u to v*. If T is omission-changeable at t_c, then T_i's deadline is changed to be the value that *would be the deadline* of a subtask with weight v released at time t_c (assuming the new deadline is smaller than the old), and furthermore all future releases of T are decreased appropriately. A two-processor example of an omission-changeable task T:$[1/6, 1/2]$ is depicted in Fig. 10(a). Notice that T_2 is released two time units after T changes its weight. This spacing is in keeping with the window length of a task with weight 1/2. Before discussing flow-changeable tasks, recall that each subtask accounts for one quantum's worth of execution (as formalized by property F2). If T is flow-changeable at t_c, then the release of T_{i+1} is changed to be one time slot after the time slot such that $f(T_i, t) \geq 1$, assuming the new weight is used in f's definition. A one-processor example of an flow-changeable task T:$[1/6, 1/2]$ is depicted in Fig. 10(b). Notice that T_2 is released at time slot 4, which is the time slot at which its total flow equals 1.

Because these two rules require that priority queues be reordered for every weight change, the time complexity of one task changing its weight is $\Theta(logN)$. Consequently, when these rules are used to change the weight of every task in the system, the time complexity is $\Theta(N logN)$. Thus, fine-grained reweighting produces small drift, but the time complexity incurred may be a concern if N is large.

Lazy reweighting. On simple method for decreasing the time complexity of fine-grained reweighting (at the cost of accuracy) is to spread out the time over which all tasks are reweighted. Under *lazy reweighting* a task changes its weight (using the fine-grained rules) the *next* time it is scheduled, after the weight change has occurred. Consider the one-processor example depicted in Fig. 11, which consists of three tasks: U:1/2, which leaves at time 2, V:$[1/4, 1/2]$, which changes weight at time 2, and W:1/4. Since the system is lazy-reweighted, V's weight change is not enacted until the first time it is scheduled after it changes weight. As a consequence, V's second subtask is not released until time 4 and V's third subtask is released at time 6. (If the system were

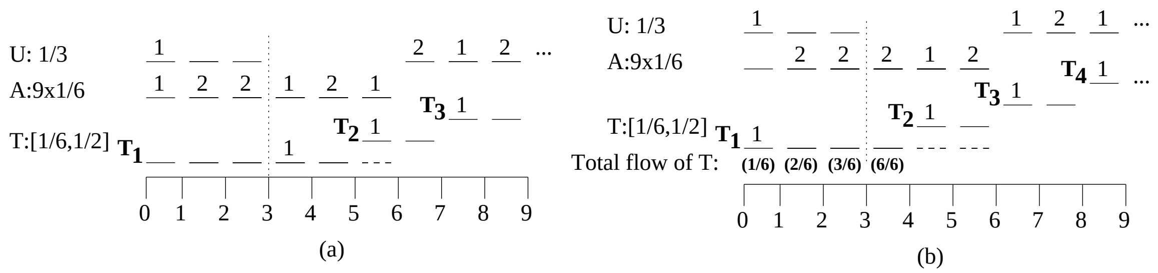

Figure 10: A two-processor system consisting of the tasks U:1/3 that leaves at time 3, T:$[1/6, 1/2]$, and a set A of nine tasks all with a minimum and maximum weight of 1/6. The dotted lines represent time slots that are originally part of T_1's window, which due to reweighting is truncated. **(a)** T has a lower priority than any task in A. Consequently, it is scheduled after time 3, and hence it is *omission-changeable*. **(b)** T has a higher priority than any task in A. Consequently, it is scheduled before time 3, and hence it is *flow-changeable*. The flow of task T for the first four time slots is given at the bottom of the inset.

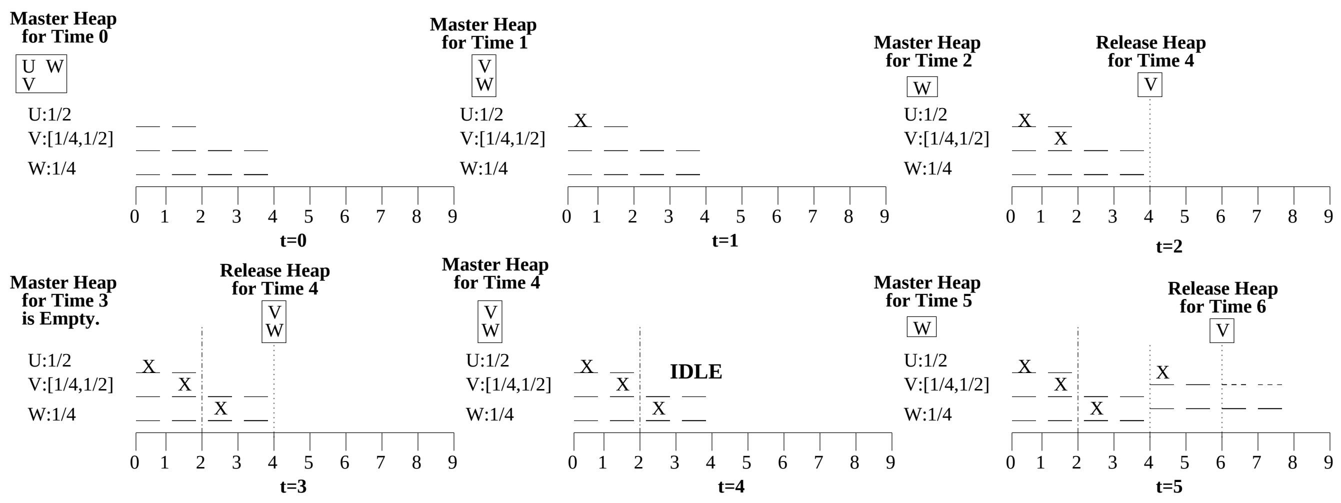

Figure 11: A one-processor example implementation of the PD^2 algorithm using lazy-reweighting. This system contains three tasks: U:1/2, which leaves at time 2, V:$[1/4, 1/2]$, and W:1/4. As before, each box above the schedule denotes either a release or master heap and the Xs denote times in which tasks are scheduled. Notice that the processor is idle in time slot $[3, 4)$.

fine-grained, then V's second subtask would be released at time 3; if the system were not reweighted, then V's third subtask would be released at time 8.) Because lazy-reweighting distributes the time over which weight changes are enacted, the time complexity for lazy reweighting is $\Theta(M\,logN)$. Since a subtask may be delayed for up to *two window lengths* (as defined by the corresponding task's old weight) before its weight change is enacted, and the drift incurred in each time slot between the reweighting event and its enactment is equal to the new weight minus the old weight, the maximal drift incurred per weight change under lazy reweighting is given by twice the new weight minus the old weight times the window length of the task prior to the weight change, which is upper bounded by $2 \cdot (maxwt(T)\text{-}minwt(T))/minwt(T)$. Notice that lazy reweighting is identical to leave/join reweighting (as described in Sec. 1), except that under lazy reweighting, a task's weight is changed via Block and Anderson's reweighting rules whereas leave/join reweighting changes a task's weight via the leave/join rules. Consequently, leave/join and lazy reweighting have the same worst-case bounds for drift and time complexity, but as we experimentally verify in Sec. 4, lazy reweighting achieves substantially better bounds on drift, in practice.

k-fine grained reweighting. Lazy reweighting and fine-grained reweighting represent two different extremes of when to reweight a task. Fine-grained reweighting changes a task's weight immediately, and lazy reweighting delays until its next release. *k-fine-grained reweighting* is a hybrid of the two methods, under which every task that is scheduled (and has not yet been reweighted) and some set of k tasks are reweighted at each time slot. We refer to the order in which these k additional tasks are chosen to be reweighted as the *k-list*. Tasks that are reweighted first are said to have a *lower ordering* on the k-list.

We propose two methods for ordering the k-list: **(i)** the importance of a task (as determined by the developer);

and **(ii)** by which task would benefit the most from a weight change. The best method for ordering the k-list under suggestion (ii) is to prioritize based on the ratio of each task's weight after the change to its weight before the change; tasks with a larger new-to-old ratio have a lower ordering in the k-list. Unfortunately, in the worst case, prioritizing tasks using this method has a time complexity of $\Omega(N\,log N)$—the very cost we are trying to avoid. The only alternative is to use heuristics. As we have discussed, under lazy reweighting, the drift associated with a reweighting event may be as high as $2 \cdot (maxwt(T)\text{-}minwt(T))/minwt(T)$. Thus, if we want to minimize the amount of drift that occurs and we have no prior knowledge of the reweighting events that are to occur, we suggest that tasks in the k-list be prioritized based on the value $(maxwt(T)\text{-}minwt(T))/minwt(T)$. Under this method, for example, a task T:$[1/8, 1/4]$ would have a lower ordering than a task Z:$[1/4, 3/8]$. Regardless of how tasks are ordered, within N/k quanta of a reweighting event, every task will be reweighted. Hence, the maximal drift that any task will incur per reweighting event under k-fine-grained reweighting is $min\big(2 \cdot (maxwt(T) - minwt(T))/minwt(T), 2 + N \cdot (maxwt(T) - minwt(T))/k\big)$. Since the k tasks with the lowest ordering will always be reweighted immediately, these tasks incur a drift of at most two quanta per reweighting event.

So far in our discussion we have implicitly assumed that if a second weight change occurs, then all tasks have enacted the first. However, for task sets in which this is not the case (*i.e.*, multiple reweighting events that occur within N/k quanta of each other), the user must choose between limiting the drift of the k tasks with the lowest ordering *or* of all tasks. If limiting the drift of the k lowest-ordered tasks is of the utmost importance, then in the scenario where two reweighting events occur within N/k quanta of each other, the k-list is reset, and the k lowest-ordered tasks immediately enact the second weight change. If limiting the maximal drift in the system is more important, then at the second reweighting, in this same scenario, the k-list is left untouched, and the next k tasks that are reweighted are the same k that would have been reweighted had the second reweighting event not occurred, thus servicing every task in a "round robin" manner.

4 Experimental Results

To empirically evaluate leave/join, fine-grained, lazy, and k-fine-grained reweighting, we constructed three different experiments, each run for 1,000 time steps. In each experiment, each task was randomly assigned a minimum weight in the range $[1/500, 1/100]$ and a maximum weight that was either one or two orders of magnitude larger than its minimum. All experiments allow share changes of up to two orders of magnitude, since, as stated in the introduction, such share changes are possible in Whisper. We refer to a task that has a maximum weight two orders of magnitude larger than its minimum as a *high-variance* task. For all experiments, the value k in k-fine-grained reweighting is defined to be the number of processors. At time zero, all tasks have a weight equal to their minimum. At time two, there is a reweighting event that causes each task T to change its weight to a value given by the formula (taken from [2])

$$minwt(T) + (maxwt(T) - minwt(T)) \cdot \frac{M - W}{X - W}, \tag{3}$$

where M is the number of processors, W is the sum of all minimum weights, and X is the sum of all maximum weights. This formula guarantees that the sum of all weights is equal to number of processors (assuming $X \geq M$).

In the first set of experiments, we set the number of tasks to be 100 and the number of processors to be 10, and varied the number of high-variance tasks from 10 to 50. Fig. 12 shows the total drift as a function of the number of high-variance tasks. (Fine-grained and k-fine grained have values approximately equal to one.) In this and subsequent graphs, each data point represents an average of 50 trials, and error bars are used to show standard deviations (though in later graphs these error bars are not visible, due to their small value). There are five important results illustrated in this graph: **(i)** leave/join reweighting incurs twice as much drift as lazy reweighting; **(ii)** as the number of high-variance tasks increases, leave/join and lazy reweighting both exhibit a gradual decease in total drift; **(iii)** the amount of drift incurred by both lazy and leave/join reweighting is substantially larger than that for either fine- or k-fine-grained reweighting; **(iv)** fine-grained reweighting incurs only *slightly* less drift than k-fine-grained reweighting; and **(v)** k-fine-grained reweighting's drift is relatively small even when the number of high-variance tasks is five times the value of k. Result (i) is a consequence of the fact that lazy reweighting enacts a weight change by changing the deadline of the *current* subtask of a task (and/or the release of its successor), whereas leave/join reweighting changes the deadline of the *next* subtask (and/or the release of the subtask after that). Result (ii) is an artifact of defining the weights in accordance with Equation (3), which decreases the amount by which a task increases its weight as the number of high-variance tasks increase. Result (iii) confirms experimentally one of analytical results of this paper. Finally, results (iv) and (v) are both a consequence of having a relatively large k-to-N ratio.

In the second set of experiments, we set the number of tasks to be 200 and the number of high-variance tasks to be equal to 50, and varied the number of processors between 5 to 25. Fig. 13 shows the number of heap operations as a function of the processor count. (The number of heap operations directly relates to the running time of a reweighting algorithm.) Though difficult to see, lazy and leave/join reweighting have an approximately-equal number of heap operations, and k-fine-grained has approximately twice the number of heap operations as lazy reweighting. Fine-grained reweighting, on the other hand, entails a substantially larger number of heap operations. Notice that, with lazy, leave/join, and k-fine-grained reweighting, the number of heap operations increases linearly with the processors (as expected). On the other hand, with fine-grained reweighting a linear

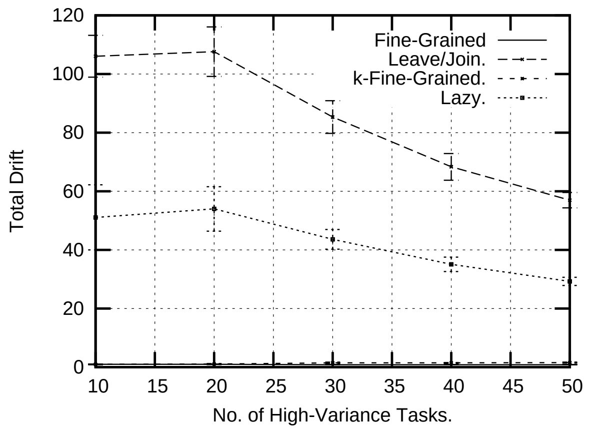

Figure 12: Varying the high-variance tasks.

decrease occurs, for reasons that we do not yet understand.

In the final set of experiments, we set the number of processors to be five and the number of high-variance tasks to be ten, and we varied the number of tasks between 50 to 250. Fig. 13 shows the number of heap operations as a function of the task count. Again, though difficult to see, lazy and leave/join reweighting have approximately equal values and k-fine-grained reweighting results in approximately twice the number of heap operations as lazy reweighting. Notice that, with lazy, leave/join, and k-fine-grained reweighting, the number of heap operations increases slowly (with values commensurate with a logarithmic increase). On the other hand, fine-grained reweighting increases by a super-linear amount (*i.e.*, commensurate with $N log N$ time complexity).

The most interesting result of these experiments is the efficacy of k-fine-grained reweighting under the metrics of running time and total drift. k-fine-grained reweighting's performance is due in large part to the relatively low ratio of the number of tasks to processors. As the ratio increases, k-fine-grained will be less able to have both a low running time and small total drift. Notice, however, that in Fig. 14, as the task-to-processor ratio increases,

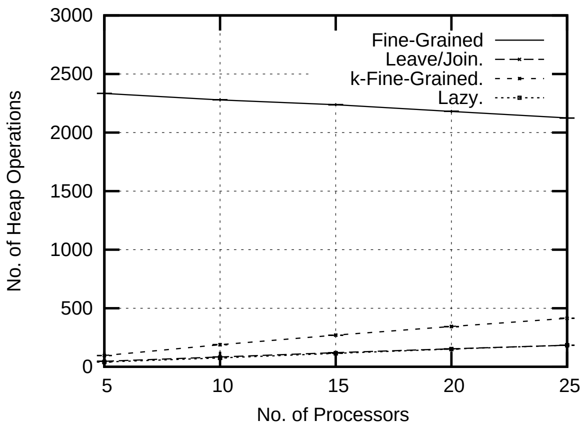

Figure 13: Varying the number of processors.

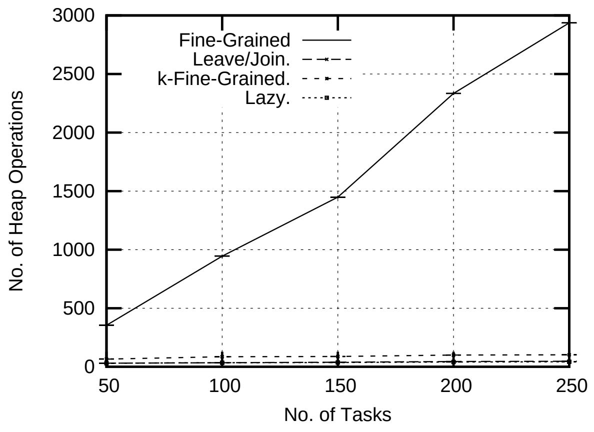

Figure 14: Varying the number of tasks.

fine-grained reweighting's running time deteriorates super-lineally, thus further justifying the use of k-fine-grained reweighting, when small amounts of error are acceptable.

5 Concluding Remarks

We have constructed two new methods for reweighting tasks that allow the user to control the natural trade-off between worst-case time complexity and drift. We have given an analytical comparison of the two reweighting schemes presented in this paper (lazy and k-fine-grained reweighting) with two previous schemes (leave/join reweighting and fine-grained reweighting) in Fig. 16. Furthermore, we have presented experiments that confirm this analysis. These experiments suggest that, in some settings, k-fine-grained reweighting may be the best choice if both running time and total drift are important.

While our focus in this paper has been scheduling techniques that *facilitate* fine-grained adaptations, techniques for determining *how* and *when* to adapt are equally important. Such techniques can either be application-specific (*e.g.*, adaptation policies unique to a tracking system like Whisper) or more generic (*e.g.*, feedback-control mechanisms incorporated within scheduling algorithms [11]). Both kinds of techniques warrant further study, especially in the domain of multiprocessor platforms.

References

[1] A. Block, and J. Anderson. Fine-grained task reweighting on multiprocessors. Submitted to *the 11th IEEE Real-time and Embedded Technology and Applications Symp.*, March. 2005.

[2] J. Anderson, A. Block, and A. Srinivasan. Quick-release fair scheduling. In *Proc. of the 24th IEEE Real-time Sys. Symp.*, pages 130–141, Dec. 2003.

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

[4] J. Anderson and A. Srinivasan. Pfair scheduling: Beyond periodic task systems. In *Proc. of the Seventh International Conference on Real-time Computing Sys. and Applications*, pages 297–306, Dec. 2000.

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

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

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

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

[9] S.J. Golestani. A self-clocked fair queueing scheme for broadband applications. In *Proc. of IEEE INFOCOM '94*, pages 636–646, April 1994.

[10] P. Goyal, H. Vin, and H. Cheng. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. In *Proc. of ACM SIGCOMM'96*, pages 157–168, Aug. 1996.

[11] C. Lu, J. Stankovic, G. Tao, and S. Son. Design and evaluation of a feedback control edf scheduling algorithm. In *Proc. of the 20th IEEE Real-time Sys. Symp.*, pages 44–53, Dec. 1999.

[12] A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical Report MIT/LCS/TR-297, Massachusetts Institute of Technology, 1983.

[13] A. K. Parekh. *A Generalized Processor Sharing Approach To Flow Control in Integrated Services Networks.* PhD thesis, MIT, 1992.

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

[15] A. Srinivasan and J. Anderson. Fair scheduling of dynamic task systems on multiprocessors. In *Proc. of the 11th International Workshop on Parallel and Distributed Real-time Sys.*, April 2003, on CD ROM.

[16] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C.G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proc. of the 17th IEEE Real-time Sys. Symp.*, pages 288–299, 1996.

[17] N. Vallidis. *WHISPER: A Spread Spectrum Approach to Occlusion in Acoustic Tracking.* Ph.D. thesis, University of North Carolina at Chapel Hill, 2002.

[18] J. Xu and R. Lipton. On fundamental tradeoffs between delay bounds and computational complexity in packet scheduling algorithms. In *Proc. of the 2002 Special Interest Group on Data Communication*, pages 19–23, August 2002.

Appendix

If an adaptable tasks T changes its weight from u to v at t_c and $u > 1/2$ (*i.e.*, T is heavy), then T is considered to be *heavy-changeable.* Note that if a task is heavy-changeable then it *cannot* be flow-changeable or omission-changeable. Let T_j be the subtask of task T with the smallest index that is released before t_c and has a deadline at or after t_c. If T is heavy-changeable at t_c, then T leaves at $d(T_j)$ and rejoins at $d(T_j) + 2$ with a new weight. Moreover, all subtask of T (and if $v < u$ all subtasks of any task that utilize this newly created capacity) that are released before $D(T_j)$ must be eligible one time slot *before* they are released. By artificially lengthening the next few subtask windows, heavy-changeable tasks are able to "short circuit" rule L. As an example, consider Fig. 15. In inset (a), a task T increases its weight from 5/7 to 6/7 at time 10. Note that the first few subtasks released after time 10 have an window length of three. Actually, a task of weight 6/7 should only have windows of length two, so this window has been artificially lengthened. In inset (b), T decreases its weight from 6/ 7 to 5/7 at time 8, and a task A of weight 1/7 joins at time 8. Notice that the first subtask of T released after 8 has a window of length three, and the first subtask of A has a window length of eight. both these window lengths are one more than if the tasks had joined "normally" at time 14.

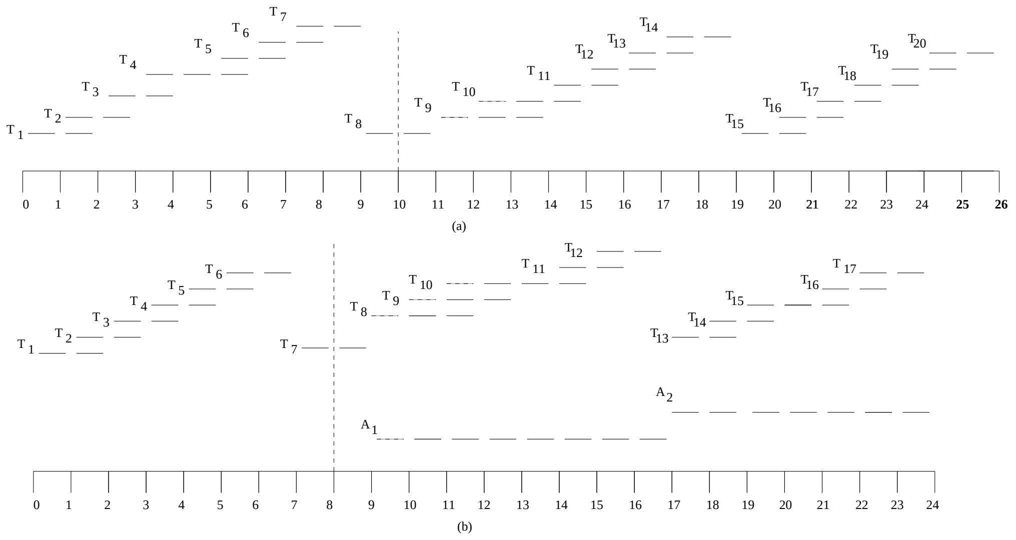

Figure 15: Two different systems demonstrating the effect of changing a heavy task. **(a)** A task T of weight 5/7 changes to 6/7 at time 8. **(b)** A task T of weight 6/7 changes to 5/7 at time 8 and a task A of weight 1/7 joins at time 8.

Name	**Time Complexity**	**Maximal Drift per Change**	**"Top k" Drift per Change**
Fine-Grained	$\Theta(N log\ N)$	5	5
k-Fine-Grained	$\Theta((M+k) log\ N)$	$min\left(\frac{2 \cdot Q(T)}{minwt(T)}, \frac{Q(T) \cdot N}{k} + 5\right)$	5
Lazy	$\Theta(M log\ N)$	$\frac{2 \cdot Q(T)}{minwt(T)}$	$\frac{2 \cdot Q(T)}{minwt(T)}$
Leave/Join	$\Theta(M log\ N)$	$\frac{2 \cdot Q(T)}{minwt(T)}$	$\frac{2Q(T)}{minwt(T)}$

Figure 16: Comparison of reweighting schemes. The function $Q(T) = maxwt(T) - minwt(T)$. With heavy tasks.

Notice that if T is a heavy-changeable task, then it releases a subtask with a window defined by its new weight at time $d(T_j) + 2$. Since a heavy tasks has a window length of at most 3, it follows that the resulting allocation error is at most five quanta. From this discussion, we have the slightly modified table.