Fine-Grained Task Reweighting on Multiprocessors*

Aaron Block, James H. Anderson, and Gary Bishop
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 as a weight. Task reweighting can be used as a means for consuming (or making available) spare processing capacity. In this paper, we propose a multiprocessor reweighting scheme that can change a task's processor share with "minimal" error per share change.

1 Introduction

Two trends are evident in recent work on real-time systems. First, *multiprocessor* designs are becoming quite 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 *fine-grained adaptivity*, *i.e.*, the ability to react to external events within short time scales by adjusting task parameters, particularly *processor shares*. Examples of such applications include human-tracking systems, computer-vision systems, and signal-processing applications such as synthetic aperture imaging.

One such application is the Whisper tracking system designed at the University of North Carolina to perform full-body tracking in virtual environments [10]. Like many tracking systems, Whisper uses *predictive techniques* to track objects. The computational cost of making the "next" prediction in tracking an object 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*. Moreover, share changes must be enacted within *time scales as short as 10 ms.*

In this paper, we focus our attention on a particular class of global scheduling algorithms known as *fair* scheduling algorithms (specifically, Pfair algorithms [3], as introduced later.) 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. Share allocations, in practical schemes, track the ideal with only bounded "error." We consider an allocation policy to be *fine-grained* if any additional per-task "error" (in comparison to the ideal allocation) caused by a task share-change request is constant; we use the term *drift* to refer to this source of error, and refer to the process of changing a task's share as *reweighting*.

Srinivasan and Anderson [8] have given sufficient conditions (described in Sec. 2) under which tasks may dynamically join and leave a running Pfair-scheduled system without causing any missed deadlines. As discussed later, these rules require that tasks sometimes be delayed when leaving the system. As a result of these "leaving delays," any reweighting scheme constructed from these rules is *coarse-grained*, *i.e.*, susceptible to non-constant drift.

In this paper, we show (for the first time) that fine-grained reweighting on multiprocessors is possible by presenting reweighting rules that ensure *constant* drift. These rules are introduced in the following way. After first presenting a more careful review of prior work in Sec. 2, we present a new task model in Sec. 3 that allows task weights to vary with time. In Sec. 4 we present our reweighting rules, and show that they ensure constant drift. It can be shown that *zero drift is not possible*; hence, our rules cannot be substantially improved. In Sec. 5, we assess the efficacy of these rules via 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 $0 < wt(T) \leq 1$. Due to page limitations, we henceforth assume $wt(T) \leq \frac{1}{2}$ for all T. Tasks of "heavier" weight require additional reasoning, as described in the full paper (found at `http://www.cs.unc.edu/~anderson/papers.html`).

Under Pfair scheduling, processor time is allocated in discrete time units, called *quanta*; the time interval $[t, t+1)$,

*Work supported by NSF grants CCR 0204312, CNS 0309825, and CNS 0408996. The first author was also supported by an NSF fellowship.

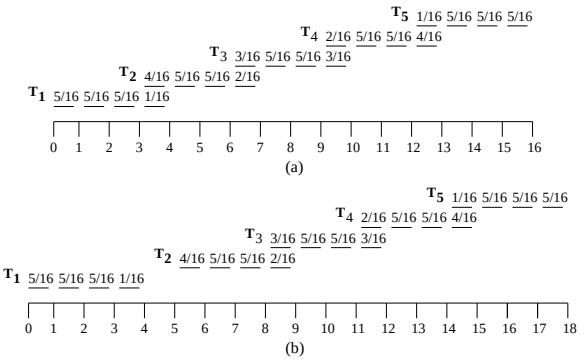

Figure 1. f for a **(a)** periodic and **(b)** IS task of weight 5/16.

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*. The function $S(T, t)$ gives the total number of slots allocated to task T in the slot interval $[0, t)$.

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 at time t is captured by the function $lag(T, t)$, which is defined as $wt(T) \cdot t - S(T, t)$. 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. These error bounds are ensured by treating each quantum of a task's execution, henceforth called a *subtask*, as a schedulable entity. The i^{th} subtask of task T, denoted T_i, where $i \geq 1$, has an associated *pseudo-release* $r(T_i) = \lfloor (i-1)/wt(T) \rfloor$ and *pseudo-deadline* $d(T_i) = \lceil i/wt(T) \rceil$. (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 $(\forall T, t :: -1 < lag(T, t) < 1)$ is maintained. In Fig. 1(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 slots 3–6. (Tasks execute sequentially, so if T_1 is scheduled in slot 3, then T_2 is scheduled in slots 4–6.)

IS model. The intra-sporadic (IS) task model [7] generalizes the well-known sporadic task model [6] by allowing subtasks to be released late. This extra flexibility is useful in many applications where processing steps may be delayed. Fig. 1(b) illustrates the Pfair windows of an IS task. An IS task T is *active* at time t if there exists a subtask T_k such that $r(T_k) \leq t < d(T_k)$, and is *passive* otherwise. In Fig. 1(b), T is active in all slots but slot 4. 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. The release and deadline of a subtask T_i of an IS task T are defined as $r(T_i) = \theta(T_i) + \lfloor (i-1)/wt(T) \rfloor$ and $d(T_i) = \theta(T_i) + \lceil i/wt(T) \rceil$, where the offsets satisfy the property $k \geq i \Rightarrow \theta(T_k) \geq \theta(T_i)$.

```
f(T_i: subtask, t: integer)
1:  if t < r(T_i) ∨ t > (d(T_i) − 1) then
2:      f(T_i, t) := 0
3:  else if t = r(T_i) then
4:      if i = 1 ∨ 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 − Σ_{q=0}^{t−1} f(T_i, q), wt(T))
11: else
12:     f(T_i, t) := wt(T)
13: fi
```

Figure 2. Pseudo-code defining $f(T_i, t)$.

PD^2. The PD^2 [7] Pfair scheduling algorithm is optimal for scheduling IS tasks on an arbitrary number of processors. It prioritizes subtasks on an earliest-pseudo-deadline-first (EPDF) basis, and uses two tie-breaking rules. For the case wherein all task weights are at most 1/2 (our focus here), PD^2 uses one 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 1 if T_i's window overlaps T_{i+1}'s, and is 0 otherwise. In Fig. 1, $b(T_i) = 1$ 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. Further ties are broken arbitrarily. (See [2] 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 [7]. Let $ideal(T, t)$ denote the share that T receives in a fluid schedule in $[0, t)$. Then, $lag(T, t) = ideal(T, t) - S(T, t)$. $ideal(T, t)$ is defined in terms of a function *flow*(T, u) that gives the share assigned to task T in slot u. *flow*(T, u) is in turn defined in terms of a function f that indicates the share assigned to each *subtask* T_i 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. 2. Observe that, while subtask T_i is active, $f(T_i, t)$ usually equals $wt(T)$, but may be less than $wt(T)$ in slots $r(T_i)$ and $d(T_i) - 1$. Some example f values are given in Fig. 1. 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. 1(a), $\sum_{T_i \in T} f(T_i, 3) = \frac{1}{16} + \frac{4}{16}$, and $\sum_t f(T_2, t) = \frac{4}{16} + \frac{5}{16} + \frac{5}{16} + \frac{1}{16} = 1$. Having defined f, *flow*(T, t) is defined as $\sum_i f(T_i, t)$. In Fig. 1(a), *flow*$(T, 3) = f(T_1, 3) + f(T_2, 3) + f(T_3, 3) + ... = \frac{1}{16} + \frac{4}{15} + 0 + ... = \frac{5}{16}$. $ideal(T, t)$ is defined as $\sum_{u=0}^{t-1}$ *flow*(T, u).

Dynamic task systems. The leave/join conditions of Srinivasan and Anderson [8] mentioned earlier, and a theorem concerning them, are stated below.

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

L: (*leave condition*) Let T_i denote the last-scheduled subtask of T. T can leave at time t iff $t \geq d(T_i) + b(T_i)$.

Theorem 1 ([8]). PD^2 *correctly schedules any feasible dynamic IS task system satisfying J and L.*

Note that a task may reweight by leaving with its old weight and rejoining with its new weight.

3 Adaptable Task Model

The *adaptable* task model proposed here extends the IS model by allowing the weight of a task T, $wt(T, t)$, to be a function of time. The notion of an ideal system considered earlier for periodic and IS tasks served two purposes. First, it gave us a means for defining "allocation error," *i.e.*, lag. Second, it gave us a means for defining subtask releases and deadlines. For adaptable tasks, it is convenient to use a different notion of an "ideal" system for each purpose. The *true ideal system*, proposed below, provides a means of accurately tracking allocation error. Its definition does not depend on how subtask releases and deadlines are defined. In contrast, the *windowed ideal system*, defined later, accounts for shares on a per-subtask basis in order to define a subtask's releases and deadlines.

True ideal system. The true ideal system can enact weight changes instantaneously, and hence is the *ultimate standard* for defining allocation error. A task T's true ideal allocation up to time t, $true\_ideal(T, t)$, is given by

$$true\_ideal(T, t) = \int_0^t wt_0(T, u)du, \qquad (1)$$

where $wt_0(T, u)$ equals $wt(T, u)$ if T is active at u, and 0 otherwise [9]. Unlike elsewhere in this paper, in (1), t is allowed to be any rational value.

Windowed ideal system. For an IS task, a subtask T_i's deadline is defined to be the time t such that at some time within $(t-1, t]$ the cumulative flow allocated to T_i equals a unit share. For an adaptable task, if a subtask T_i's deadline were determined using its cumulative *true ideal* instead, then *no* earliest-pseudo-deadline-first (EPDF) algorithm (such as PD^2) could correctly schedule all task sets. This follows from a counterexample given in the full paper, which is omitted here due to space constraints.

The windowed ideal system determines each subtask's releases and deadlines so that the drift per weight change is *constrained to a small constant value*. Since the windowed ideal system defines the behavior of adaptable tasks, we omit the term "windowed" where this does not cause confusion. As definitions fundamental to the (windowed) ideal system are introduced, the one-processor system in Fig. 3 is used as an example. This system consists of a task U of weight 1/2 and a task T with a weight of 5/16 that increases to 1/2 at time 7. Inset (a) depicts true ideal allocations and $true\_ideal(T, t)$, and inset (b) shows the f values (defined below) for all three tasks and $ideal(T, t)$ (defined below). (f and $ideal(T, t)$ define the behavior of adaptable tasks.)

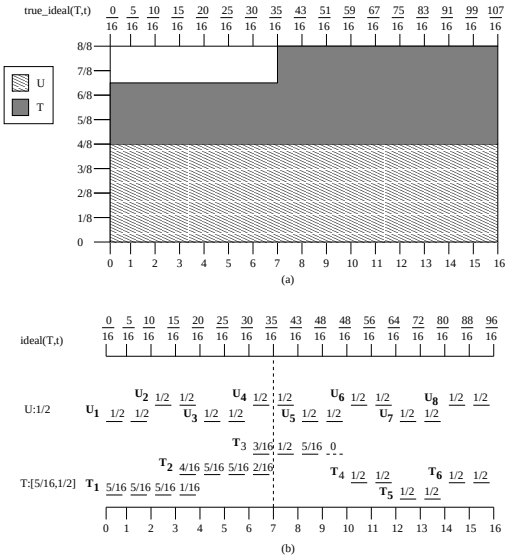

Figure 3. Both insets depict a one-processor system consisting of a task U of weight 1/2 and a task T with a weight of 5/16 that increases to 1/2 at time 7. **(a)** True ideal allocations and $true\_ideal(T, t)$. **(b)** The windows and f values for all three tasks and $ideal(T, t)$.

In defining an adaptable subtask's deadlines and releases, it is useful to define two different notions of a subtask's deadline. A subtask T_i's *scheduling deadline* $d(T_i)$ (defined later) is used to determine T_i's priority for scheduling and cannot change once it has been set. A subtask T_i's *flow-based deadline* $fd(T_i)$ is defined to be the time t such that $\sum_{q<t} f(T_i, q) = 1$, where $f(T_i, t)$ is a "windowed version" of per-slot flow for subtask T_i at t.

As with IS tasks, f can be defined mathematically, but we opt instead for a pseudo-code-based definition, shown in Fig. 4. There are three changes in f's new definition: in lines 5, 7, 10, and 11, $wt(T, t)$ is used instead of $wt(T)$; in lines 1, 7, and 9, $fd(T_i)$ is used instead of $d(T_i)$; and in line 4, $f(T_i, t) := wt(T, t)$ if $d(T_{i-1}) \neq fd(T_{i-1})$. These changes account for T's time-varying weight. As before, properties F1 and F2 hold for this new definition.

Given the above definition of f, *flow* and *ideal* can be defined, as before. In fact, the definition of $flow(T, t)$ is exactly the same as before, except that the new definition of f is used. A task T's ideal allocation up to (integral) time t,

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

Figure 4. Pseudo-code defining the new $f(T_i, t)$

ideal(T, t), is given by the formula

$$\mathit{ideal}(T, t) = \int_0^t \mathit{flow}(T, u)du = \sum_{u=0}^{t} \mathit{flow}(T, u).$$

As before, T's lag is given by $\mathit{lag}(T, t) = \mathit{ideal}(T, t) - S(T, t)$. To see the difference between *ideal*(T, t) and *true_ideal*(T, t), consider Fig. 3. Notice that until time 8, T's windowed ideal allocation is the same as its true ideal allocation; however, at time 9 the windowed ideal is 3/16 less then the true ideal, and at time 11 the windowed ideal is 11/16 less than the true ideal. Moreover, after time 11 the windowed ideal is always 11/16 less than the true ideal (though both now increase at a rate of 1/2).

Releases and scheduling deadlines. The *projected flow-based deadline of subtask T_i at u*, *pfd*(T_i, u), is the time that *would* be the flow-based deadline of subtask T_i *if* T's weight remained static after time u. In Fig. 3(b), *pfd*$(T_1, 0) = 4$, *pfd*$(T_3, 6) = 10$, and *pfd*$(T_3, 7) = 9$.

The *scheduling deadline* (or the *deadline*, for short) of a subtask T_i is defined as $d(T_i) = \mathit{pfd}(T_i, r(T_i))$.

Let $R(T_i)$ denote the earliest time T_i can be released (as defined by the application being scheduled). Then, $r(T_i)$ is defined as follows: for $i = 1$, $r(T_1) = R(T_1)$, and for $i > 1$, $r(T_i) = max(R(T_i), d(T_{i-1}) - b(T_{i-1}), \mathit{fd}(T_{i-1}))$. In Fig. 3(b), $r(T_1) = 0$, $r(T_3) = 6$, and $r(T_4) = 10$.

4 Fine-Grained Reweighting

Fine-grained reweighting improves upon coarse-grained reweighting by changing future subtask releases and deadlines. Let t_c denote a time at which one or more tasks are to be reweighted. We can guarantee fine-grained reweighting by applying one or more of the rules specified below. The choice of which rule to apply depends on whether the "currently active" subtask of a task has been scheduled by t_c. (We remind the reader that all weights are assumed to be at most 1/2; as explained in the full paper, an additional rule is required if this is not the case.) Let T_j be the subtask of task T with the smallest index such that $r(T_j) < t_c \leq d(T_j)$. Let $u = \mathit{wt}(T, t_c - 1)$ and let v be T's new weight. For example, in Fig. 5(a), t_c = 10, $T_j = T_2$, $u = 3/20$, and $v = 1/2$. We say that T is *flow-changeable at time t_c from weight u to v* if T_j is scheduled before t_c, and otherwise is *omission-changeable at time t_c from u to v*.

Rule O: If T is omission-changeable at time t_c from u to v, then it leaves with weight u at time *max*$(d(T_{j-1}) + b(T_{j-1}), t_c)$ and immediately rejoins with weight v.

Rule F: If T is flow-changeable at time t_c from u to v, then it leaves with weight u at time *min*(*fd*(T_j), $d(T_j)) + b(T_j)$ and instantly rejoins with weight v.

The terms "leaving" and "joining" should be viewed as conceptional notions since it is both easier to comprehend and define a weight-varying task as a series of short-lived tasks, each with one weight, rather than one long-running task with multiple weights. Note that weight decreases do not free capacity until the change is completed. Both rules are extensions of the leave/join rules L and J given earlier in Sec. 2. However, the rules above exploit the specific circumstances that occur when a task changes its weight to "short circuit" rules L and J, so that reweighting is accomplished faster. By rule L, T can leave at time $d(T_k) + b(T_k)$, where T_k is its last-scheduled subtask. If task T (as defined above) is omission-changeable, then its subtask T_j has not been scheduled by time t_c. Such a task can be viewed as having left the system at time *max*$(d(T_{j-1}) + b(T_{j-1}), t_c)$, in which case, it can rejoin the system immediately at time t_c. In Fig. 5(b), task T of weight 3/20 leaves at time 8 and task U of weight 1/2 joins at time 10. In Fig. 5(c), task T increases its weight from 3/20 to 1/2 via rule O. Note that in Fig. 5(b), T behaves as if it leaves at time 8 and rejoins at time 10 with its new weight.

If T is flow-changeable, then by rule L, it may leave at time $d(T_j) + b(T_j)$. However, if *fd*$(T_j) + b(T_j)$ occurs earlier, then T may leave then. ("Flow-changeable" refers to the fact that a task's flow-based deadline may be used in changing its weight.) Since both the ideal and actual systems have finished executing T_j by time *fd*(T_j), the proof that leaving at this time cannot cause deadline misses is virtually identical to that of rule L. In Fig. 5(d), task T increases its weight from 3/20 to 1/2 at time 10 via rule F. Note that T "leaves" at its flow-based deadline at time 12, which is two time units earlier than its scheduling deadline.

OF-reweighting (respectively, *LJ-reweighting*) refers to reweighting via rules O and F (resp., the leave/join rules L and J) under PD^2. Since O and F are extensions of L and J, the following theorem follows from Theorem 1.

Theorem 2. *Under OF-reweighting, no scheduling deadline is missed, provided the sum of all task weights is at most M (the number of processors) at any time.*

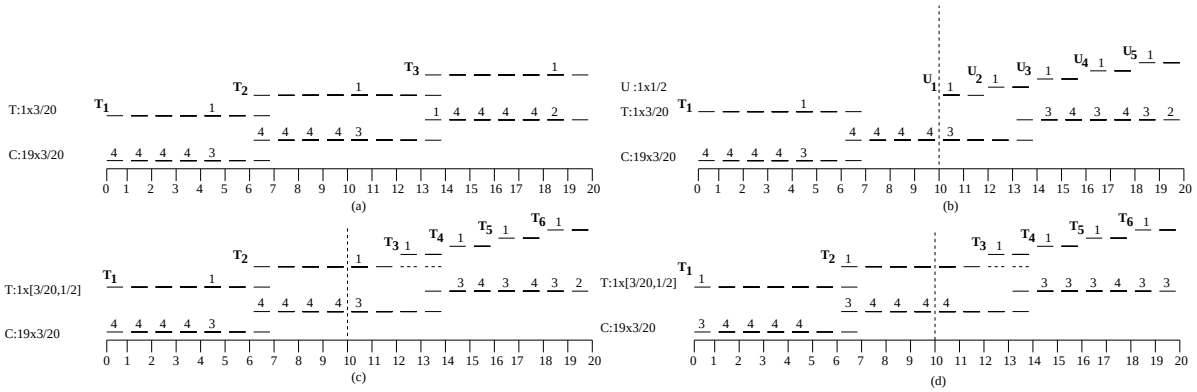

Figure 5. A four-processor system consisting of a set C of 19 tasks of weight 3/20 each, and a task T of weight 3/20, and in (b), a task U of weight 1/2. In (a)–(c), all ties are broken in favor of tasks from C, and in (d), all ties are broken in favor of task T. The numbers in each slot denote the number of tasks from each set scheduled in that slot. **(a)** No weight change. **(b)** T leaves at time 8 and U joins at time 10. **(c)** T reweights to 1/2 via rule O at slot 10. **(d)** T reweights to 1/2 via rule F at slot 10. Notice that rule O or F is applied depending on whether T_2 is scheduled prior to slot 10.

Drift. To show that OF-reweighting is fine-grained, we must show that the drift (from the *true* ideal) is constant. Towards that end, we introduce the concept of *total drift*. For a given allocation policy $\mathcal{A}$, total drift is defined 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 policy $\mathcal{A}$ for task T up to time t, and $wt_0(T,u)$ is as defined earlier in Sec. 3. Given this terminology, a reweighting algorithm $\mathcal{A}$ is *fine-grained* iff there exists a constant c such that for all times t and tasks T, $-c \cdot n < t\_drift_{\mathcal{A}}(T,t) < c \cdot n$ holds, where n is the number of times that T has changed its weight in the interval $[0,t]$. Note that a given reweighting policy $\mathcal{A}$ is not fine-grained if for an *arbitrary* value c, there exists a system that contains a task T that changes its weight once at time t such that $t\_drift(T,t) \geq c$.

Before showing that OF-reweighting is fine-grained, we first prove that LJ-reweighting is not. Consider the four-processor system depicted in Fig. 6(b). The depicted system consists of a set A of 35 tasks with weight 1/10 and a task T with weight 1/10 that increases to 1/2 at time 4. In this example, task T changes its weight via LJ-reweighting. The weight change is not enacted until time 10. As a consequence, T's total drift reaches a value of 5 at time 10. This example can be generalized to generate any value of total drift for T, by decreasing its initial weight. Under LJ-reweighting, such a task cannot change its weight until the end of the first window generated by its initial weight. Hence, by decreasing the weight of T to $1/(2(c+1))$, we have $t\_drift_{\mathcal{LJ}}(T, d(T_1)) \geq c$. The theorem below follows.

Theorem 3. *LJ-reweighting is not fine-grained.*

Next, consider OF-reweighting. By Theorem 2, to prove that OF-reweighting is fine-grained, we merely need to consider the window placements of a task after it is reweighted. Suppose a task T's weight is changed at time t_c and T_j is its active subtask at t_c. If T changes its weight via rule O, then the resulting allocation error is clearly at most one quantum. In Fig. 5(b), the true ideal flow in slots 6 through 9 associated with the "omitted" subtask T_2 is "lost." This total flow is at most one quantum. If T changes its weight via rule F, then it releases a subtask with a window defined by its new weight at time $min(fd(T_j), d(T_j)) + b(T_j)$. Under a true ideal allocation, T changes to its new weight within the quantum $[fd(T_j)-1, fd(T_j))$. It follows that the resulting allocation error is at most one quantum. From this discussion, we have the following theorem.

Theorem 4. *OF-reweighting is fine-grained.*

5 Experimental Results

In this section we present a set of experiments that involve randomly-generated task sets. Such experiments are of interest because they allow a wide range of system configurations to be considered. (In the full version of this paper, we present an additional set of experiments that involve a simulation of Whisper itself.)

In these experiments, each task T has a *minimum weight*, $minwt(T)$, and a *maximum weight*, $maxwt(T)$, such that $minwt(T) \leq wt(T,t) \leq maxwt(T)$ holds for all t. Tasks were randomly assigned a minimum weight in the range $[1/500, 1/100]$ and a maximum weight that is either two times or two orders of magnitude larger. We refer to a task that has a maximum weight two orders of magnitude larger than its minimum as a *high-variance* task. High-variance tasks were included because share changes of two orders of magnitude can occur in Whisper. All simulations were run for 1,000 time steps, and repeated 61 times. In each graph presented below, 98% confidence intervals are given. The simulated platform was assumed to be similar to a testbed

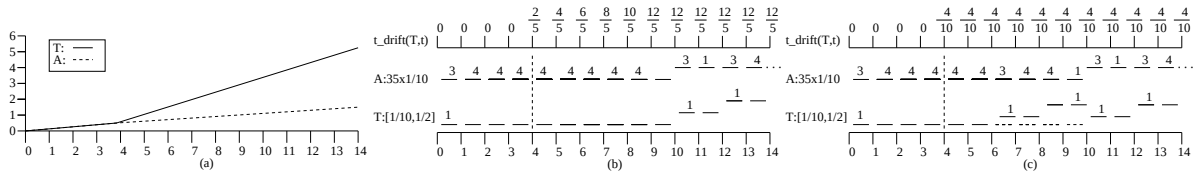

Figure 6. A four-processor system with a set A of 35 tasks with weight 1/10 and a task T with weight 1/10 that increases to 1/2 at time 4. The total drift for T is labeled in (b) and (c). **(a)** The ideal allocations for each task. **(b)** LJ-reweighting. **(c)** OF-reweighting.

system in our lab that is comprised of 2.7 GHz processors. On this testbed, we implemented and timed both reweighting algorithms considered in our simulations and found that all per-slot scheduling decisions could be made in at most 5.7 μs for the largest task systems considered in our experiments. Assuming a 1ms scheduling quantum, we regarded this value as insignificant and thus ignored scheduling overheads in our simulations. In each experiment, each task T was initially assigned a weight equal to its minimum and then was reweighted at time 500 so that its new weight is given by the formula (taken from [1])

$$minwt(T) + (maxwt(T) - minwt(T)) \cdot \frac{M - W}{X - W}, \quad (2)$$

where M is the number of processors, W is the sum of all minimum weights, and X is the sum of all maximum weights. Assuming $X \geq M$, (2) guarantees that the sum of all weights equals the number of processors.

We conducted experiments in which the number of high-variance tasks varied from 0 to 50, the number of processors from 4 to 16, and the number of tasks from 50 to 200. However, due to space limitations, the graphs below present only a representative sampling of the data that we collected.

In the first set of graphs, in insets (a)–(b) of Fig. 7, the number of tasks is 50 and the number of processors is set at either 4 or 16. Inset (a) shows, for a 4-processor systems, the maximal drift of any task, and the average drift of all tasks, at time 1,000, as a function of the number of high-variance tasks. Inset (b) gives the total amount of computation completed by time 1,000, as a percentage of the ideal allocation, as a function of the number of high-variance tasks, for both 4 and 16 processors. In inset (a), the OF-Max and OF-Avg curves are difficult to distinguish from the x-axis since the largest maximal drift incurred by OF-reweighting was 0.923. For LJ-reweighting, on the other hand, the largest maximal and average drift incurred were 75.8 and 6.1, respectively. Furthermore in inset (b), OF-reweighting stabilizes at approximately 100% (99.4%) of the ideal allocation for 4 (16) processors. In contrast, LJ-reweighting stabilizes at approximately 85% (83%) for 4 (16) processors. Notice that, for both 4- and 16-processor systems, *with only two high-variance tasks and only one weight change*, LJ-reweighting is only within 90% of the ideal, while OF-reweighting is virtually at 100%.

In inset (b), OF-reweighting actually exceeds the ideal allocation for some task sets. This is because the system is underutilized before time 500, providing an opportunity for both OF- and LJ-reweighting to be *slightly* ahead of the ideal system. Since a task under OF-reweighting is not heavily penalized for changing its weight, this "over-allocation" carries over to time 1,000. OF-reweighting will only be slightly ahead of the ideal because of Pfair restrictions. Notice that, while the maximal drift drops as the number of high-variance tasks increases, the average drift increases and the percentage of the ideal decreases. Thus, while the "worst" task may perform better, the system on the whole performs worse.

In insets (c)–(d) of Fig. 7, the number of high-variance tasks is 10, the total number of tasks is either 50 or 200, and the number of processors varies from 4 to 16. Inset (c) shows the maximal drift of any task, and the average drift of all tasks, at time 1,000, for task sets with 50 tasks. Inset (d) shows the total amount of computation completed by time 1,000 as a percentage of the ideal allocation for task sets of size 50 and 200. Notice that, for LJ-reweighting, the amount of computation completed stabilizes at approximately 85% (89%) of the ideal for systems with 50 (200) tasks. In contrast, for OF-reweighting, the amount of computation completed stablizes at approximately 100% (101%) for systems with 50 (200) tasks.

In all of these experiments, OF-reweighting substantially outperformed LJ-reweighting. In scenarios where a task's weight increases by two orders of magnitude, at least one task under LJ-reweighting incured two orders of magnitude of drift. Furthermore, Fig. 7(a) illustrates that, even when the maximal drift of any task is as small as 11, LJ-reweighting completes as little as 85% of the ideal allocation 500 time steps after a weight change, in systems with *only one weight change*.

6 Concluding Remarks

We have shown (for the first time) that fine-grained reweighting is possible on multiprocessor platforms. The experiments reported herein show that our reweighting rules enable greater precision in adapting than LJ-reweighting. However, this added precision comes at the price of higher scheduling costs. $\Omega(max(N, M \log N))$ time is *required* to reweight N tasks simultaneously. In contrast, LJ-reweighting entails only $O(M \log N)$ time. However, as

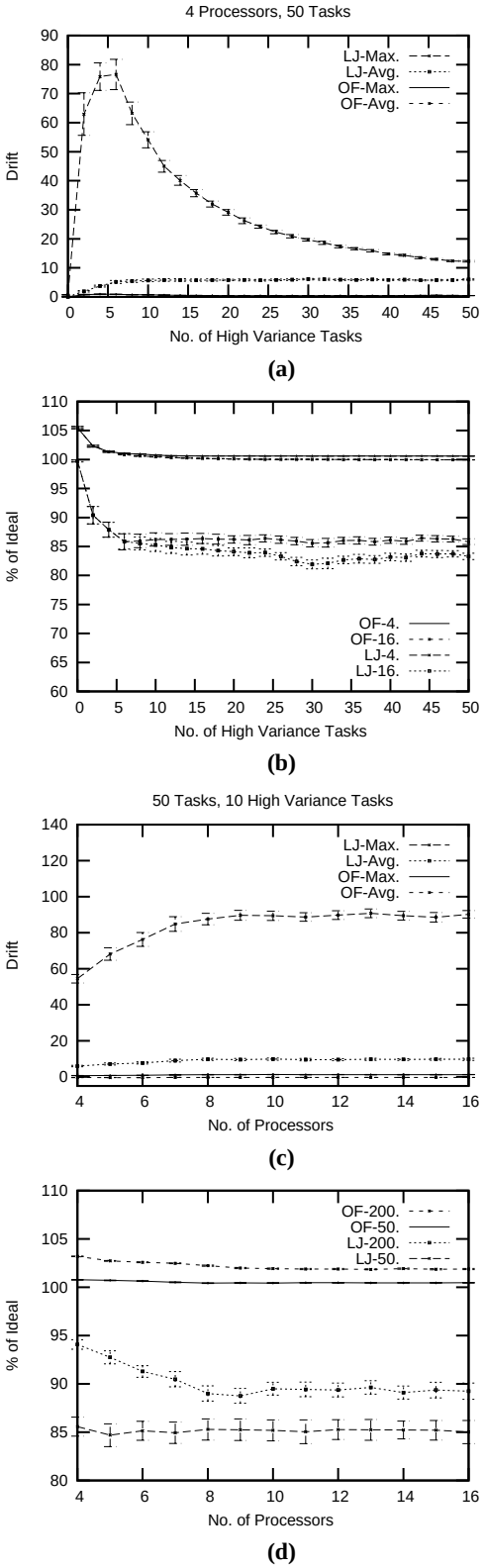

Figure 7. Selection of experiments. For clarity, the legend in each inset orders the curves in the (top-to-bottom) order they appear in that graph. If curves cannot be observed, it is because they are indistinguishable from the x-axis.

noted earlier, experiments conducted on our testbed system indicate that scheduling overheads will likely be small in practice under either scheme. Moreover, we have shown in a related paper that this precision-versus-overhead tradeoff can be balanced by using schemes that are hybrids of "pure" OF- and LJ-reweighting [4].

As mentioned earlier, 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 [5]). Both kinds of techniques warrant further study, especially in the domain of multiprocessor platforms.

References

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

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

[3] S. Baruah, J. Gehrke, and C.G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proc. of the 9th Int'l Parallel Processing Symp.*, pp. 280–288, 1995.

[4] A. Block and J. Anderson. Task reweighting on multiprocessors: Efficiency versus accuracy. In *Proc. of the 13th Int'l Workshop on Parallel and Distributed Real-time Sys.*, 2005, on CD ROM.

[5] 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.*, pp. 44–53, 1999.

[6] A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical Report MIT/LCS/TR-297, M.I.T., 1983.

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

[8] A. Srinivasan and J. Anderson. Fair scheduling of dynamic task sys. on multiprocessors. In *Proc. of the 11th Int'l Workshop on Parallel and Distributed Real-time Sys.*, 2003, on CD ROM.

[9] 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 sys. In *Proc. of the 17th IEEE Real-time Sys. Symp.*, pp. 288–299.,1996.

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