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

{block, anderson, gb}@cs.unc.edu

Abstract

We consider the problem of *task reweighting* in fair-scheduled multiprocessor systems wherein each task's processor share is specified as a *weight*. When a task is reweighted, a new weight is computed for it, which 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. In prior work on *uniprocessor* notions of fairness, a number of fine-grained reweighting schemes were proposed. However, in the multiprocessor case, prior work has failed to produce such a scheme. In this paper, we remedy this shortcoming by presenting a multiprocessor reweighting scheme that is fine-grained. We also present an experimental evaluation of this scheme that shows that it is often much more responsive than prior (non-fine-grained) schemes in enacting weight-change requests.

Keywords: Adaptive, multiprocessor, Pfair, reweighting

[*]Work supported by NSF grants CCR 0204312, CNS 0309825, CNS 0408996, and CCF 0541056. The first author was also supported by an NSF fellowship. A preliminary version of this paper appeared in *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 429-435, August 2005

1 Introduction

Two trends are evident in recent work on real-time systems. First, *multiprocessor* designs are becoming 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.

To better motivate the need for fine-grained adaptivity, we consider in this paper one particular example application in some detail—namely, the Whisper tracking system designed at the University of North Carolina to perform full-body tracking in virtual environments [14]. 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 consider the specific issue of how to support adaptive behavior such as this on (tightly-coupled) identical multiprocessors when using *fair* global scheduling algorithms, specifically Pfair algorithms [3], as introduced later. In fair scheduling schemes, 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. In practical schemes, share allocations track the ideal scheduler with only bounded "error." We consider an allocation policy to be *fine-grained* if any additional per-task "error" (in comparison to an 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 [12] 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. These rules can be applied to reweight tasks: such a task simply leaves with its old weight and rejoins with its new weight. However, as discussed later, these rules require that tasks sometimes be delayed when leaving the system. Because of these "leaving delays," any reweighting scheme constructed from these rules is *coarse-grained*, *i.e.*, susceptible to non-constant drift.

After the presentation of the results herein in preliminary form [6], we subsequently considered the use of partitioned [4] and global earliest-deadline-first (EDF) [8] scheduling algorithms to schedule highly-adaptive symmetric multiprocessor workloads. While partitioning and global EDF provide excellent average-case performance and can reduce migration and preemption costs associated with Pfair scheduling, both algorithms have substantial drawbacks. Specifically, under partitioning, fine-grained reweighting is (provably) impossible; under global EDF, fine-grained reweighting is possible only if deadline misses are permissible.

In this paper, we show that fine-grained reweighting (without deadline misses) is possible under Pfair scheduling 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 in Sec. 3 a new task model that allows task weights to vary with time and associated reweighting rules. In Sec. 4, we prove that, under our reweighting rules, no task misses a deadline and drift is constant. (The proof that deadline misses are avoided is rather lengthy; much of this proof is deferred to an expanded version of this paper, which is available as a technical report [7].) We also show 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 involving the Whisper system.

2 Preliminaries

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.) In this paper, all time values are assumed to indicate an integral number of quanta, unless specified otherwise. Throughout the paper, we use M to denote the number processors in the system.

As mentioned in the introduction, under Pfair scheduling, correctness is defined by comparing to an "ideal" scheduling algorithm that can guarantee each task precisely its required share over any time interval. Thus, as we introduce increasingly flexible task models, we also introduce increasingly more general (and complex) notions of ideal scheduling. The behavior of each scheduling algorithm (ideal or otherwise) presented in this paper is defined by the sequence of its allocation decisions over time. The total time allocated to a task T in an arbitrary schedule $\mathcal{S}$ over the range $[t_1, t_2)$ is denoted as $\mathsf{A}(\mathcal{S}, T, t_1, t_2)$. Similarly, we use $\mathsf{A}(\mathcal{S}, T_j, t_1, t_2)$ and $\mathsf{A}(\mathcal{S}, \tau, t_1, t_2)$ to denote, respectively, the total allocations to the "subtask" T_j (as defined below) and to all tasks in the set τ over the range $[t_1, t_2)$. As a shorthand, we use $\mathsf{A}(\mathcal{S}, T, t)$ (resp., $\mathsf{A}(\mathcal{S}, T_j, t)$) to denote $\mathsf{A}(\mathcal{S}, T, t, t+1)$ (resp., $\mathsf{A}(\mathcal{S}, T_j, t, t+1)$), and we say that T is *scheduled* at time t if $\mathsf{A}(\mathcal{S}, T, t) = 1$. In order to compare the difference between the allocations to a task T up to time t in two schedules $\mathcal{S}$ and $\mathcal{I}$, where $\mathcal{S}$ is an "actual" schedule and $\mathcal{I}$ is an "ideal" one, we use the function $\mathsf{lag}(\mathcal{S}, \mathcal{I}, T, t) = \mathsf{A}(\mathcal{I}, T, 0, t) - \mathsf{A}(\mathcal{S}, T, 0, t)$. Additionally, we use the function $\mathsf{LAG}(\mathcal{S}, \mathcal{I}, \tau, t) = \sum_{T \in \mathcal{T}} \mathsf{lag}(\mathcal{S}, \mathcal{I}, T, t)$ to compare the differences in allocations for *all tasks* in the task set τ in schedules $\mathcal{S}$ and $\mathcal{I}$. We assume $\mathsf{lag}(\mathcal{S}, \mathcal{I}, T, 0) = 0$. Thus, $\mathsf{LAG}(\mathcal{S}, \mathcal{I}, \tau, t)$ can be rewritten as

$$\mathsf{LAG}(\mathcal{S}, \mathcal{I}, \tau, t) = \mathsf{LAG}(\mathcal{S}, \mathcal{I}, \tau, t-1) + (\mathsf{A}(\mathcal{I}, \tau, t-1) - \mathsf{A}(\mathcal{S}, \tau, t-1)). \tag{1}$$

For brevity, we denote $\mathsf{lag}(\mathcal{S}, \mathcal{I}, T, t)$ as $\mathsf{lag}(T, t)$ and $\mathsf{LAG}(\mathcal{S}, \mathcal{I}, \tau, t)$ as $\mathsf{LAG}(\tau, t)$, when $\mathcal{S}$ and $\mathcal{I}$ are well-defined and obvious. (Examples illustrating the concepts in this paragraph are given shortly.)

Periodic Pfair scheduling. 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* (or *utilization*) $\mathsf{wt}(T) = T.e/T.p$, where $0 < \mathsf{wt}(T) \leq 1$. The period of a task defines both the relative deadline of each of its job and the allowed separation between jobs: successive jobs of a periodic (sporadic) task T are released exactly (at least) $T.p$ time units apart. The execution cost of a task is the amount of time for which each job of that task must be scheduled. Due to page limitations, we focus exclusively in this paper on tasks with weight at most $1/2$. Tasks of weight greater than $1/2$, called *heavy tasks*, require additional reasoning, which can be found in the first author's forthcoming Ph.D. dissertation. (It is worth noting that the Whisper system used as a test case herein requires task weights of at most 1/3.)

The ideal schedule for a periodic task system allocates $\mathsf{wt}(T)$ processing time to each task in each time slot. Thus, for a periodic system τ, $\mathsf{lag}(T,t)$ is defined as $\mathsf{wt}(T) \cdot t - \mathsf{A}(\mathcal{S}, T, 0, t)$, where $\mathcal{S}$ is some "real" schedule of τ. The schedule $\mathcal{S}$ is *Pfair* iff $(\forall T \in \tau, t :: -1 < \mathsf{lag}(T,t) < 1)$. 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. Scheduling decisions are made only at quantum boundaries. The i^{th} subtask of task T, denoted T_i, where $i \geq 1$, has an associated *pseudo-release* $\mathsf{r}(T_i) = \lfloor (i-1)/\mathsf{wt}(T) \rfloor$ and *pseudo-deadline* $\mathsf{d}(T_i) = \lceil i/\mathsf{wt}(T) \rceil$. (For brevity, we often drop the prefix "pseudo-.") It can be shown that if each subtask T_i is scheduled in the interval $\mathsf{w}(T_i) = [\mathsf{r}(T_i), \mathsf{d}(T_i))$, termed its *window*, then $(\forall T \in \tau, t :: -1 < \mathsf{lag}(T,t) < 1)$ is maintained [2]. As an example, in Fig. 1(a), $\mathsf{r}(T_2) = 3$, $\mathsf{d}(T_2) = 7$, and $\mathsf{w}(T_2) = [3,7)$. (This figure also depicts per-slot ideal allocations for each subtask, 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 must be scheduled in slots 4–6.)

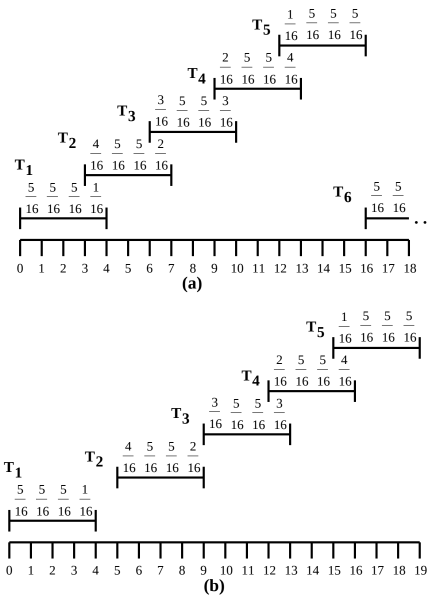


Figure 1: $\mathsf{A}(\mathcal{I}, T_j, t)$ for a **(a)** periodic and **(b)** IS task of weight 5/16. The windows of successive subtasks are as indicated (*e.g.*, T_1's window in both insets is $[0,4)$).

IS model. The *intra-sporadic* (*IS*) *task model* [11] generalizes the well-known sporadic task model [10] 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 of weight 5/16 in which the release of T_2 is delayed by two quanta and the release of T_3 is delayed by an additional quantum. 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. For example, in Fig. 1(b), $\theta(T_1) = 0$, $\theta(T_2) = 2$, and for $i \geq 3$, $\theta(T_i) = 3$. The release and deadline of a subtask T_i of an IS task T are defined as $\mathsf{r}(T_i) = \theta(T_i) + \lfloor (i-1)/\mathsf{wt}(T) \rfloor$ and $\mathsf{d}(T_i) = \theta(T_i) + \lceil i/\mathsf{wt}(T) \rceil$, where the offsets satisfy the property $k \geq i \Rightarrow \theta(T_k) \geq \theta(T_i)$. A subtask T_i is *active* at time t iff $\mathsf{r}(T_i) \leq t < \mathsf{d}(T_i)$, and a task T is *active* at t iff it has an active subtask at t. For example, in Fig. 1(b), T is active in every slot except slot 4. If $\theta(T_i) < \theta(T_{i+1})$, then we say that there is an *IS separation* between T_i and T_{i+1}. (Note that an extension of the IS model exists in which a subtask T_i can become eligible before $\mathsf{r}(T_i)$ [11]. All the results of this paper can be easily extended to such a model, but for clarity, we do not consider this extension to the IS model.)

The PD^2 algorithm. The PD^2 Pfair scheduling algorithm [11] 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, $\mathsf{b}(T_i)$, which is defined as $\lceil i/\mathsf{wt}(T) \rceil - \lfloor i/\mathsf{wt}(T) \rfloor$. In a periodic task system, $\mathsf{b}(T_i)$ is 1 if T_i's window overlaps T_{i+1}'s, and is 0 otherwise. For example, in each inset in

Fig. 1, $\mathsf{b}(T_i) = 1$ for $1 \leq i \leq 4$ and $\mathsf{b}(T_5) = 0$. 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 an explanation of this tie-breaking rule.) Notice that, in the absence of IS separations, $\mathsf{r}(T_{i+1}) = \mathsf{d}(T_i) - \mathsf{b}(T_i)$. For example, in Fig. 1(a), $\mathsf{r}(T_2) = \mathsf{d}(T_1) - \mathsf{b}(T_1) = 4 - 1 = 3$, and $\mathsf{r}(T_6) = \mathsf{d}(T_5) - \mathsf{b}(T_5) = 16 - 0 = 16$.

IS ideal schedule. Ideal allocations within the IS task model can be defined in much the same way as for periodic tasks [11]; however, we must modify this definition to allow for IS separations. Before continuing, notice that since the total allocation to a task in a given time slot equals the total allocation to all of its subtasks in that slot, $\mathsf{A}(\mathcal{S}, T, t) = \sum_{T_j \in T} \mathsf{A}(\mathcal{S}, T_j, t)$. For example, in Fig. 1(a), $\mathsf{A}(\mathcal{I}, T, 6) = \mathsf{A}(\mathcal{I}, T_1, 6) + \mathsf{A}(\mathcal{I}, T_2, 6) + \mathsf{A}(\mathcal{I}, T_3, 6) + ... = 0 + 2/16 + 3/16 + 0 + ... = 5/16$. Thus, per-task and per-task-set allocations in a schedule $\mathcal{S}$ over an arbitrary interval can be defined by simply defining $\mathsf{A}(\mathcal{S}, T_j, t)$ for an arbitrary subtask T_j and time slot t.

For an arbitrary IS task system τ, we let $\mathcal{I}_{\text{IS}}$ denote the ideal schedule of τ. $\mathsf{A}(\mathcal{I}_{\text{IS}}, T_j, t)$ can be defined using an arithmetic expression, but we have opted instead for a more intuitive pseudo-code-based definition in Fig. 2. The ideal IS schedule allocates each subtask T_j some amount of processing time in each slot of its window. For slots other than $\mathsf{r}(T_i)$ and $\mathsf{d}(T_i) - 1$, this allocation is $\mathsf{wt}(T)$. T_i's allocation in slots $\mathsf{r}(T_i)$ and $\mathsf{d}(T_i) - 1$ are adjusted so that **(i)** T_i's entire allocation (across all slots in its window) is one, and **(ii)** T_i's allocation in slot $\mathsf{r}(T_i)$ (resp., $\mathsf{d}(T_i) - 1$) plus T_{i-1}'s (resp., T_{i+1}'s) allocation in slot $\mathsf{d}(T_{i-1}) - 1$ (resp., $\mathsf{r}(T_{i+1})$) equals $\mathsf{wt}(T)$ (assuming those subtasks exist). Examples of such allocations are given in Fig. 1.

```
$\mathsf{A}(\mathcal{I}_{\text{IS}}, T_i, t)$
1:  if $t < \mathsf{r}(T_i) \vee t \geq \mathsf{d}(T_i))$ then
2:      $\mathsf{A}(\mathcal{I}_{\text{IS}}, T_i, t) := 0$
3:  else if $t = \mathsf{r}(T_i)$ then
4:      if $i = 1 \vee \mathsf{b}(T_{i-1}) = 0$ then
5:          $\mathsf{A}(\mathcal{I}_{\text{IS}}, T_i, t) := \mathsf{wt}(T)$
6:      else
7:          $\mathsf{A}(\mathcal{I}_{\text{IS}}, T_i, t) :=$
                $\mathsf{wt}(T) - \mathsf{A}(\mathcal{I}_{\text{IS}}, T_{i-1}, \mathsf{d}(T_{i-1}) - 1)$
8:      fi
9:  else
10:     $\mathsf{A}(\mathcal{I}_{\text{IS}}, T_i, t) :=$
            $\min(\mathsf{wt}(T), 1 - \mathsf{A}(\mathcal{I}_{\text{IS}}, T_i, 0, t))$
11: fi
```

Figure 2: Pseudo-code defining $\mathsf{A}(\mathcal{I}_{\text{IS}}, T_i, t)$.

Dynamic task systems. The *dynamic IS* task model an extension of the IS model in which tasks can leave and join by the aforementioned conditions by Srinivasan and Anderson [12], which 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.

L: (*leave condition*) A task T can leave at time t iff either it has not been scheduled prior to t, or $t \geq \mathsf{d}(T_i) + \mathsf{b}(T_i)$ holds, where T_i is its last-scheduled subtask prior to t.

For example, in Fig. 1(b), if T were to leave after T_1, then T could not leave until time 5 because $5 = \mathsf{d}(T_1) + \mathsf{b}(T_1) = 4 + 1$. On the other hand, if T were to leave after T_5, then T could not leave until time 19 because $19 = \mathsf{d}(T_5) + \mathsf{b}(T_5) = 19 + 0$.

Theorem 1 ([12]). PD^2 *correctly schedules any dynamic IS task system satisfying J and L.*

By Theorem 1, a task T can be reweighted by leaving with its old weight (at a time that satisfies the leave condition, which may be after the reweighting event occurs) and then rejoining with its new weight (provided the join condition holds).

3 Adaptable Task Model and Fine-Grained Reweighting

In this section, we introduce the *adaptable IS* (*AIS*) task model, and define corresponding fine-grained reweighting rules, which allow the PD^2 algorithm to schedule each subtask without missing a deadline and to ensure constant drift per weight change. The AIS task model is an extension of IS task model, where the weight of each task T, $\mathsf{wt}(T, t)$, is a function of time t. (Note that, in this paper, we do concern ourselves with how task weights are determined. Such weights could be determined, for example, based upon application-specific criteria, feedback information, *etc.*)

3.1 Adaptable Task Model

A task T *changes weight* or *reweights* at time $t+1$ if $\mathsf{wt}(T, t) \neq \mathsf{wt}(T, t+1)$. If a task T changes weight at a time t_c between the release and the deadline of some subtask T_j, then the following three actions *may* occur: **(i)** if T_j has not been scheduled by t_c, then T_j may be "halted" at t_c; **(ii)** $\mathsf{r}(T_{j+1})$ *may* be redefined to be less than $\mathsf{d}(T_j) - \mathsf{b}(T_j)$; and **(iii)** if (ii) holds, then the windows of T_j and T_{j+1} may overlap by more than $\mathsf{b}(T_j)$ time slots. (In the IS model defined earlier, every subtask's deadline is at most $\mathsf{b}(T_j)$ time slots after its successor's release.)

The reweighting rules we present at the end of this section state the conditions under which the above actions may occur and the number of slots before $\mathsf{d}(T_j) - \mathsf{b}(T_j)$ that subtask T_{j+1} can be released. If T_j is *halted* before it is scheduled, then it is never scheduled. (Note that a subtask can only be halted if it has not yet been scheduled in the PD^2 schedule.) For example, consider Fig. 3(a), which depicts a task T that increases its weight from $3/19$ to $2/5$ at time 8. (The per-slot allocations and the terms "enacted" and "complete" mentioned in the figure are discussed shortly.) In this inset, T_2 halts at time 2, but for $j \in \{1, 3, 4, 5\}$, T_j does not halt. Since a subtask is only halted as a result of a reweighting event, if we do not have *a priori* knowledge of such events, then we cannot determine whether a released subtask will be halted in the future. For example, in Fig. 3(a), we have no knowledge when T_2 is released at time 6 that it will be halted at time 8.

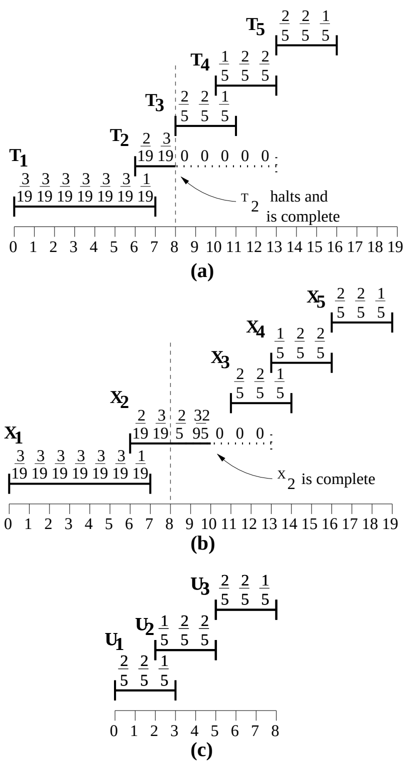


Figure 3: Per-task allocations for a task in an AIS system for **(a)** a task T with an initial weight of $3/19$ that enacts a weight change to $2/5$ and halts at time 8; **(b)** a task X with an initial weight of $3/19$ that enacts a weight change to $2/5$ at time 8 *but does not halt*; **(c)** a periodic task U with weight $2/5$. The dotted window lines indicate that the window that would have existed if the subtask task did not reweight.

Definition 1 (Initiated and Enacted). When a task reweights, there can be a difference between when it "initiates" the change and when the change is "enacted." The time at which the change is *initiated* is a user-defined time; the time at which the change is *enacted* is determined by a set of conditions dictated by the reweighting algorithm. We use the *scheduling weight of a task T at time t*, denoted $\mathsf{swt}(T, t)$, to represent the "last enacted weight of T." Formally, $\mathsf{swt}(T, t)$ equals $\mathsf{wt}(T, u)$, where u is the last time at or before t that a weight change was enacted for T. (We assume an initial weight change occurred for T where it initially joined the system.) It is important to note that, *henceforth, we compute subtask deadlines and releases using scheduling weights.* We use $\mathsf{En}(T, t)$ to denote the last time at or before time t that T enacted a weight change, and $\mathsf{Id}(T_j)$ to denote the smallest index k such that $\mathsf{En}(T, \mathsf{r}(T_j)) \leq \mathsf{r}(T_k)$. For example, in Fig. 3(b), $\mathsf{En}(X, t) = 0$, for $0 \leq t < 8$; for $j \in \{1, 2\}$, $\mathsf{Id}(X_j) = 1$; for $t \geq 8$, $\mathsf{En}(X, t) = 8$; and for $j \in \{3, 4, 5\}$, $\mathsf{Id}(X_j) = 3$. Note that if $\mathsf{Id}(T_j) = j$, then T_j is the first subtask of T released after a weight change for T has been enacted.

Definition 2 (Complete). If $\mathcal{S}$ is a schedule for the task system τ, then a subtask T_j of $T \in \tau$ is said to have *completed by time t in $\mathcal{S}$* iff $t \geq \mathsf{r}(T_j)$ and one of the following holds: **(i)** T_j has been allocated one quantum by t in $\mathcal{S}$; or **(ii)** T_j is halted by time t. As an example, consider the schedule depicted in Fig. 4, which depicts a one-processor PD^2 schedule of two tasks, T, with weight 2/5, and U, with an initial weight of $2/5$ that increases to $1/2$ at time 3 by halting U_2. In this example, T_1 completes at time 1 because it is scheduled in slot 0, whereas U_1 does not complete until time 2 because it is not scheduled until slot 1. Notice that, since U_2 is halted at time 3, it is complete at time 3 even though it is never scheduled. We use the function $\mathcal{D}(\mathcal{S}, T_j)$ to denote the (integral) time at which T_j is complete in $\mathcal{S}$.

For an adaptable task, the *deadline*, *b-bit*, and *release* of a subtask T_j, respectively, are defined by Eqns. (2)–(4), where $z = \mathsf{Id}(T_j) - 1$, $\theta(T_{j+1}) \geq \theta(T_j)$, and $\mathsf{r}(T_1)$ equals the time that T joins the system.

$$\mathsf{d}(T_j) = \mathsf{r}(T_j) + \left\lceil \frac{j - z}{\mathsf{swt}(T, \mathsf{r}(T_j))} \right\rceil - \left\lfloor \frac{j - z - 1}{\mathsf{swt}(T, \mathsf{r}(T_j))} \right\rfloor \quad (2)$$

$$\mathsf{b}(T_j) = \left\lceil \frac{j - z}{\mathsf{swt}(T, \mathsf{r}(T_j))} \right\rceil - \left\lfloor \frac{j - z}{\mathsf{swt}(T, \mathsf{r}(T_j))} \right\rfloor \quad (3)$$

$$\mathsf{r}(T_{j+1}) = \mathsf{d}(T_j) - \mathsf{b}(T_j) + (\theta(T_{j+1}) - \theta(T_j)) \quad (4)$$

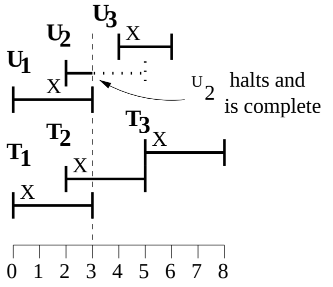


Figure 4: A one-processor PD^2 schedule for two tasks, T, with a weight $2/5$, and U, which has an initial weight of $2/5$ that increases to $1/2$ at time 3 by halting U_2. The X's denote where each subtask is scheduled.

It is important to note that since reweighting events may change a subtask's release time, Eqn. (4) *holds for the subtask T_{j+1} only if a weight change is not enacted for T over the range* $(\mathsf{r}(T_j), \mathsf{d}(T_j)]$. The above equations differ from the earlier definitions of releases, deadlines, and b-bits (given in Sec. 2) in two ways. First, (2) and (3) define the deadline and b-bit of a subtask based on the *scheduling weight* of the task at the time the subtask is *released*. Second, after a task enacts a weight change, its release, deadline, and b-bit are defined as though a new task with the new weight joined the system. (Recall that a subtask T_j is the first-released subtask after a weight change is enacted iff $\mathsf{Id}(T_j) = j$.) For example, in Fig. 3(a), after T changes its weight to $2/5$, the subtasks T_3–T_5 have similar releases, deadlines, and b-bits as the first three subtasks of the task U with weight 2/5 in inset (c).

Ideal schedules. In order to state and prove that the reweighting algorithm that we present at the end of this section does not schedule a subtask after its deadline and that it has constant drift, we introduce three notions of an ideal schedule for an AIS task system. The first ideal schedule, $\mathcal{I}_{\mathrm{SW}}$ (used for stating the reweighting rules), allocates each task a share based on its *scheduling weight* in each time slot. The second ideal schedule, $\mathcal{I}_{\mathrm{CSW}}$ (used for proving the reweighting rules and drift bounds), is the same as $\mathcal{I}_{\mathrm{SW}}$ except that $\mathcal{I}_{\mathrm{CSW}}$ is "clairvoyant" so that it does not allocate capacity to tasks that will halt. The third ideal schedule, $\mathcal{I}_{\mathrm{PS}}$ (used for proving drift bounds), allocates each task a share based on its *actual weight* in each time slot. We now formally define the allocations to a subtask in $\mathcal{I}_{\text{SW}}$ and $\mathcal{I}_{\text{CSW}}$; $\mathcal{I}_{\text{PS}}$ is considered in the next section.

As with IS tasks, $\mathsf{A}(\mathcal{I}_{\text{SW}}, T_i, t)$ can be defined mathematically, but we opt instead for a pseudo-code-based definition, shown in Fig. 5. There are three differences between the definitions of $\mathsf{A}(\mathcal{I}_{\text{IS}}, T_i, t)$ and $\mathsf{A}(\mathcal{I}_{\text{SW}}, T_i, t)$: in lines 5, 7, and 10, $\mathsf{swt}(T, t)$ is used instead of $\mathsf{wt}(T)$; and in lines 1 and 7, $\mathcal{D}(\mathcal{I}_{\text{SW}}, T_i)$ is used instead of $\mathsf{d}(T_i)$. These two changes account for T's time-varying weight. The final change is that, in line 4, $i = \mathsf{Id}(T_i)$ is used instead of $i = 1$. This change causes the per-slot allocations to T_z, where $z = \mathsf{Id}(T_i)$, to equal that of a task that joins the system at $\mathsf{r}(T_z)$. For example, in Fig. 3(a), since $3 = \mathsf{Id}(T_3)$, by lines 4 and 5, $\mathsf{A}(\mathcal{I}_{\text{SW}}, T_3, \mathsf{r}(T_3)) = \mathsf{swt}(T, \mathsf{r}(T_3)) = 2/5$, which is the same per-slot allocation that U_1 in Fig. 3(c) receives at time $\mathsf{r}(U_1)$. Before continuing, there are two important issues to note. First, in the absence of reweighting events, $\mathcal{D}(\mathcal{I}_{\text{SW}}, T_j) = \mathsf{d}(T_j)$. Second, when a task is halted via the reweighting rules given below, it is halted in both the PD^2 schedule and $\mathcal{I}_{\text{SW}}$. Since $\mathcal{I}_{\text{SW}}$ is not clairvoyant, it will allocate "normally" to a subtask until that subtask halts, after which the subtask's per-slot allocations are zero, as with T_2 in Fig. 3(a). Also note that in Fig. 3(b), X_2 is complete at time 10, since $\mathsf{A}(\mathcal{I}_{\text{SW}}, X_2, 0, 10) = 1$. Several examples of $\mathcal{I}_{\text{SW}}$ allocations are given in Fig. 3. Using the definition of $\mathcal{I}_{\text{SW}}$, we can simply define $\mathcal{I}_{\text{CSW}}$ as follows: $\mathsf{A}(\mathcal{I}_{\text{CSW}}, T_i, t) = \mathsf{A}(\mathcal{I}_{\text{SW}}, T_i, t)$, if T_i never halts, and $\mathsf{A}(\mathcal{I}_{\text{CSW}}, T_i, t) = 0$, otherwise. For example, in Fig. 3(a) $\mathsf{A}(\mathcal{I}_{\text{SW}}, T_i, t) = \mathsf{A}(\mathcal{I}_{\text{CSW}}, T_i, t)$ except for T_2, where $\mathsf{A}(\mathcal{I}_{\text{CSW}}, T_2, t) = 0$ for all t.

```
A($\mathcal{I}_{\text{SW}}, T_i, t$)
1:  if $t < \mathsf{r}(T_i) \vee t \geq \mathcal{D}(\mathcal{I}_{\text{SW}}, T_i)$ then
2:      $\mathsf{A}(\mathcal{I}_{\text{SW}}, T_i, t) := 0$
3:  else if $t = \mathsf{r}(T_i)$ then
4:      if $i = \mathsf{Id}(T_i) \vee \mathsf{b}(T_{i-1}) = 0$ then
5:          $\mathsf{A}(\mathcal{I}_{\text{SW}}, T_i, t) := \mathsf{swt}(T, t)$
6:      else
7:          $\mathsf{A}(\mathcal{I}_{\text{SW}}, T_i, t) := \mathsf{swt}(T, t) -$
                $\mathsf{A}(\mathcal{I}_{\text{SW}}, T_{i-1}, \mathcal{D}(\mathcal{I}_{\text{SW}}, T_{i-1}) - 1)$
8:      fi
9:  else
10:     $\mathsf{A}(\mathcal{I}_{\text{SW}}, T_i, t) :=$
            $\mathsf{min}(\mathsf{swt}(T, t), 1 - \mathsf{A}(\mathcal{I}_{\text{SW}}, T_i, 0, t))$
11: fi
```

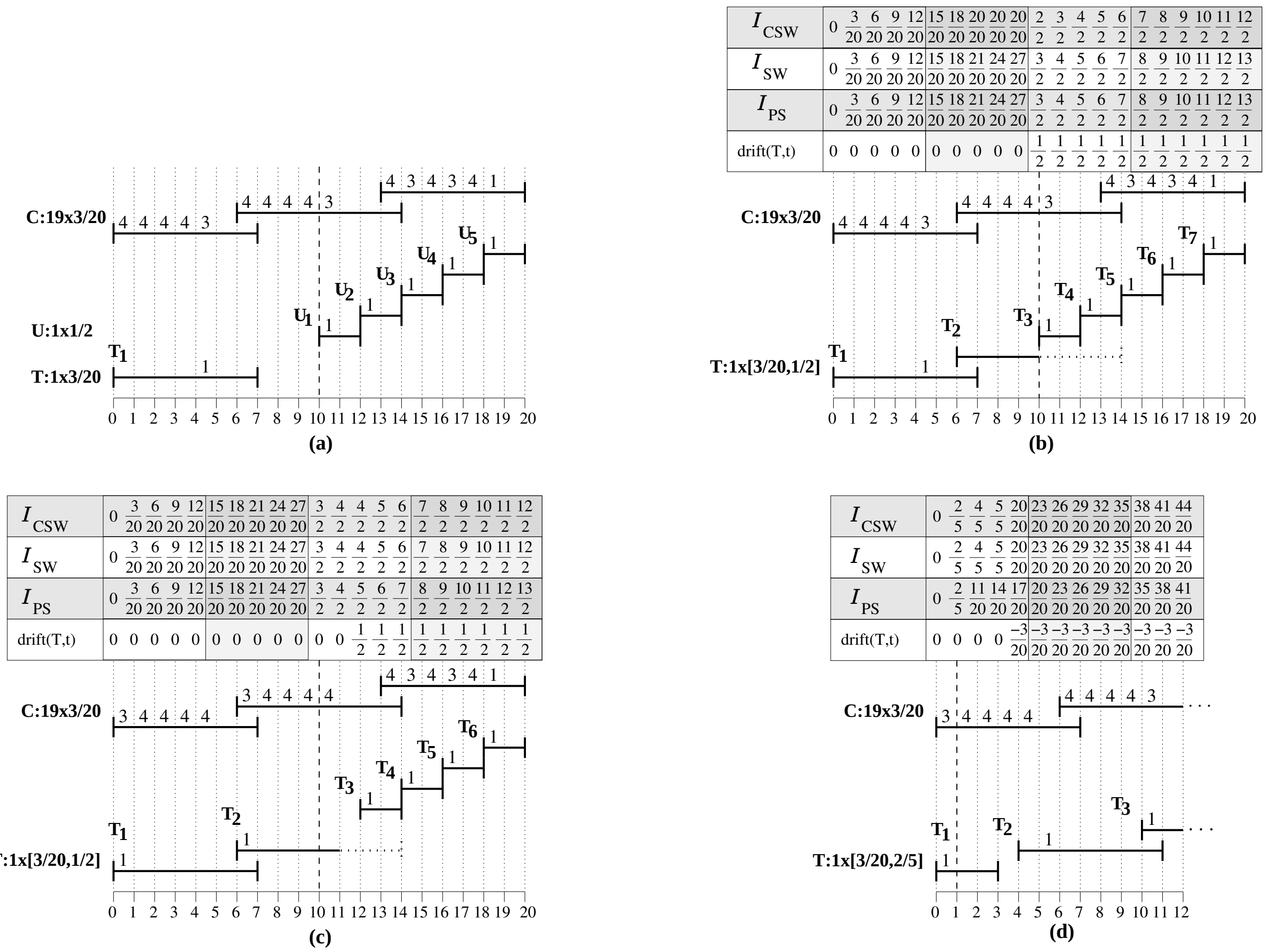
Figure 5: Pseudo-code defining the $\mathsf{A}(\mathcal{I}_{\text{SW}}, T_i, t)$.

3.2 Reweighting Rules

We now introduce two new fine-grained reweighting rules that improve upon coarse-grained reweighting by changing future subtask releases. It is important to note that in the following rules, for a given subtask T_j, the value $\mathsf{d}(T_j)$ is used to determine the scheduling priority of T_j in the PD^2 algorithm and *does not change* once T_j has been released. Furthermore, $\mathcal{D}(\mathcal{I}_{\text{SW}}, T_j)$ is used to determine the *release time* of T_j's successor, T_{j+1}. As mentioned earlier, the completion time of a subtask cannot be accurately predicted without *a priori* knowledge of weight changes; however, in the reweighting rules below, the completion time of a subtask in $\mathcal{I}_{\text{SW}}$ is only used *after* the subtask has completed, and therefore it is well-defined.

Let τ be a task system in which some task T initiates a weight change from weight w to weight v at time t_c. If there does not exist a subtask T_j of T such that $\mathsf{r}(T_j) \leq t_c$, then the weight change is enacted immediately; otherwise, let T_j denote the last-released subtask of T. If $\mathsf{d}(T_j) \leq t_c$, then the weight change is enacted at time $\mathsf{max}(t_c, \mathsf{d}(T_j) + \mathsf{b}(T_j))$. In the following rules, we consider the remaining possibility, *i.e.*, that T_j exists and $\mathsf{r}(T_j) \leq t_c < \mathsf{d}(T_j)$. For simplicity, we assume that the first subtask after a weight change by the corresponding task is released as early as possible. This assumption can be removed at the cost of more complex notation.

The choice of which rule to apply depends on whether T_j has been scheduled by t_c. We say that T is *ideal-changeable at time t_c from weight w to v* if T_j is scheduled before t_c, and otherwise is *omission-changeable at time t_c from w to v*. Because T initiates its weight change at t_c, $\mathsf{wt}(T, t_c) = v$ holds; however, T's scheduling weight does not change until the weight


C:19x3/20
U:1x1/2
T:1x3/20
U_1 U_2 U_3 U_4 U_5 T_1
T:1x[3/20,1/2]
T:1x[3/20,2/5]
T_1 T_2 T_3 T_4 T_5 T_6 T_7
(a)
(b)
(c)
(d)

(b)

I_{CSW}	0	$\frac{3}{20}$	$\frac{6}{20}$	$\frac{9}{20}$	$\frac{12}{20}$	$\frac{15}{20}$	$\frac{18}{20}$	$\frac{20}{20}$	$\frac{20}{20}$	$\frac{20}{20}$	$\frac{2}{2}$	$\frac{3}{2}$	$\frac{4}{2}$	$\frac{5}{2}$	$\frac{6}{2}$	$\frac{7}{2}$	$\frac{8}{2}$	$\frac{9}{2}$	$\frac{10}{2}$	$\frac{11}{2}$	$\frac{12}{2}$
I_{SW}	0	$\frac{3}{20}$	$\frac{6}{20}$	$\frac{9}{20}$	$\frac{12}{20}$	$\frac{15}{20}$	$\frac{18}{20}$	$\frac{21}{20}$	$\frac{24}{20}$	$\frac{27}{20}$	$\frac{3}{2}$	$\frac{4}{2}$	$\frac{5}{2}$	$\frac{6}{2}$	$\frac{7}{2}$	$\frac{8}{2}$	$\frac{9}{2}$	$\frac{10}{2}$	$\frac{11}{2}$	$\frac{12}{2}$	$\frac{13}{2}$
I_{PS}	0	$\frac{3}{20}$	$\frac{6}{20}$	$\frac{9}{20}$	$\frac{12}{20}$	$\frac{15}{20}$	$\frac{18}{20}$	$\frac{21}{20}$	$\frac{24}{20}$	$\frac{27}{20}$	$\frac{3}{2}$	$\frac{4}{2}$	$\frac{5}{2}$	$\frac{6}{2}$	$\frac{7}{2}$	$\frac{8}{2}$	$\frac{9}{2}$	$\frac{10}{2}$	$\frac{11}{2}$	$\frac{12}{2}$	$\frac{13}{2}$
drift(T,t)	0	0	0	0	0	0	0	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

(c)

I_{CSW}	0	$\frac{3}{20}$	$\frac{6}{20}$	$\frac{9}{20}$	$\frac{12}{20}$	$\frac{15}{20}$	$\frac{18}{20}$	$\frac{21}{20}$	$\frac{24}{20}$	$\frac{27}{20}$	$\frac{3}{2}$	$\frac{4}{2}$	$\frac{4}{2}$	$\frac{5}{2}$	$\frac{6}{2}$	$\frac{7}{2}$	$\frac{8}{2}$	$\frac{9}{2}$	$\frac{10}{2}$	$\frac{11}{2}$	$\frac{12}{2}$
I_{SW}	0	$\frac{3}{20}$	$\frac{6}{20}$	$\frac{9}{20}$	$\frac{12}{20}$	$\frac{15}{20}$	$\frac{18}{20}$	$\frac{21}{20}$	$\frac{24}{20}$	$\frac{27}{20}$	$\frac{3}{2}$	$\frac{4}{2}$	$\frac{4}{2}$	$\frac{5}{2}$	$\frac{6}{2}$	$\frac{7}{2}$	$\frac{8}{2}$	$\frac{9}{2}$	$\frac{10}{2}$	$\frac{11}{2}$	$\frac{12}{2}$
I_{PS}	0	$\frac{3}{20}$	$\frac{6}{20}$	$\frac{9}{20}$	$\frac{12}{20}$	$\frac{15}{20}$	$\frac{18}{20}$	$\frac{21}{20}$	$\frac{24}{20}$	$\frac{27}{20}$	$\frac{3}{2}$	$\frac{4}{2}$	$\frac{5}{2}$	$\frac{6}{2}$	$\frac{7}{2}$	$\frac{8}{2}$	$\frac{9}{2}$	$\frac{10}{2}$	$\frac{11}{2}$	$\frac{12}{2}$	$\frac{13}{2}$
drift(T,t)	0	0	0	0	0	0	0	0	0	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

(d)

I_{CSW}	0	$\frac{2}{5}$	$\frac{4}{5}$	$\frac{5}{5}$	$\frac{20}{20}$	$\frac{23}{20}$	$\frac{26}{20}$	$\frac{29}{20}$	$\frac{32}{20}$	$\frac{35}{20}$	$\frac{38}{20}$	$\frac{41}{20}$	$\frac{44}{20}$
I_{SW}	0	$\frac{2}{5}$	$\frac{4}{5}$	$\frac{5}{5}$	$\frac{20}{20}$	$\frac{23}{20}$	$\frac{26}{20}$	$\frac{29}{20}$	$\frac{32}{20}$	$\frac{35}{20}$	$\frac{38}{20}$	$\frac{41}{20}$	$\frac{44}{20}$
I_{PS}	0	$\frac{2}{5}$	$\frac{11}{20}$	$\frac{14}{20}$	$\frac{17}{20}$	$\frac{20}{20}$	$\frac{23}{20}$	$\frac{26}{20}$	$\frac{29}{20}$	$\frac{32}{20}$	$\frac{35}{20}$	$\frac{38}{20}$	$\frac{41}{20}$
drift(T,t)	0	0	0	0	$\frac{-3}{20}$	$\frac{-3}{20}$	$\frac{-3}{20}$	$\frac{-3}{20}$	$\frac{-3}{20}$	$\frac{-3}{20}$	$\frac{-3}{20}$	$\frac{-3}{20}$	$\frac{-3}{20}$

Figure 6: 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 (a), a task U of weight 1/2. In (a) and (b), all ties are broken in favor of tasks from C, and in (c), all ties are broken in favor of task T. The notation T:$[u, v]$ denotes that T's weight ranges over $[u, v]$. The windows of the various task groups are shown together. The numbers in each slot in a window denote the number of tasks from each set scheduled in that slot. (Similar notation is used in later figures.) In insets (b)–(d), T's allocation up to time t in $\mathcal{I}_{\text{SW}}$, $\mathcal{I}_{\text{CSW}}$, and $\mathcal{I}_{\text{PS}}$ as well as T's drift are labeled at the top. **(a)** T leaves at time 8 and U joins at time 10. **(b)** T reweights to 1/2 via rule O at slot 10. (Notice that T_2 is halted at t_c and is never scheduled.) **(c)** T reweights to 1/2 via rule I at slot 10. **(d)** T has an initial weight of $2/5$ that decreases to $3/20$ via rule I at time 1. Rule O or I is applied depending on whether T_2 (in (b) and (c)) or T_1 (in (d)) is scheduled prior to slot 10 (in (b) and (c)) or 1 (in (d)).

change has been *enacted*, as specified in the rules below. Note that, if t_c occurs between the initiation and enaction of a previous reweighting event of T, then the previous event is skipped, *i.e.*, treated as if it had not occurred. As discussed later, any "error" associated with skipping a reweighting event like this is accounted for when determining drift.

Rule O: If T is omission-changeable at time t_c from weight w to v and $j > 1$, then at time t_c, subtask T_j is halted and at time $\max(t_c, \mathcal{D}(\mathcal{I}_{\text{SW}}, T_{j-1}) + \mathsf{b}(T_{j-1}))$, T's weight change is enacted, and a new subtask is released. If $j = 1$, then at time t_c, T_j is halted, T's weight change is enacted, and a new subtask is released.

Rule I: If T is ideal-changeable at time t_c from weight w to v, then one of two actions is taken: **(i)** if $v > w$, then the weight change is *immediately* enacted, and at time $\mathcal{D}(\mathcal{I}_{\text{SW}}, T_j) + \mathsf{b}(T_j)$, a new subtask is released for T; **(ii)** otherwise, the weight change is enacted at time $\mathcal{D}(\mathcal{I}_{\text{SW}}, T_j) + \mathsf{b}(T_j)$, at which time a new subtask is released.

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 $\mathsf{d}(T_k) + \mathsf{b}(T_k)$, where T_k is its last-scheduled subtask. We can easily extend rule L to show that T can leave at time $\mathcal{D}(\mathcal{I}_{\mathrm{SW}}, T_k) + \mathsf{b}(T_k)$. 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 $\max(t_c, \mathcal{D}(\mathcal{I}_{\mathrm{SW}}, T_{j-1}) + \mathsf{b}(T_{j-1}))$, in which case, it can rejoin the system immediately. For example, in Fig. 6(a), task T of weight 3/20 leaves at time 8 and task U of weight 1/2 joins at time 10. In Fig. 6(b), task T increases its weight from 3/20 to 1/2 via rule O. Note that, in Fig. 6(b), T behaves as if it leaves at time 8 and rejoins at time 10 with its new weight.

If T is ideal-changeable, then by rule L, it may "leave and rejoin" with a new weight at time $\mathsf{d}(T_j) + \mathsf{b}(T_j)$ (*i.e.*, its weight change can be enacted at $\mathsf{d}(T_j) + \mathsf{b}(T_j)$). However, if $\mathcal{D}(\mathcal{I}_{\mathrm{SW}}, T_j) < \mathsf{d}(T_j)$, then T may "leave and rejoin" with a new weight at $\mathcal{D}(\mathcal{I}_{\mathrm{SW}}, T_j) + \mathsf{b}(T_j)$. ("Ideal-changeable" refers to the fact that the time at which the subtask can leave and rejoin is based on that subtask's allocations in an ideal schedule.) For example, in Fig. 6(c), task T increases its weight from 3/20 to 1/2 at time 10 via rule I. Since at time 11 the total allocation to T_2 in $\mathcal{I}_{\mathrm{SW}}$ is one, $\mathcal{D}(\mathcal{I}_{\mathrm{SW}}, T_2) = 11$. Hence, by rule I, T can "leave" at time $\mathcal{D}(\mathcal{I}_{\mathrm{SW}}, T_2) + \mathsf{b}(T_2) = 12$, which is two time units earlier than its deadline. In Fig. 6(d), task T decreases its weight from $2/5$ to $3/20$ at time one. Since T decreases its weight, by rule I, this weight change is not enacted until time $\mathcal{D}(\mathcal{I}_{\mathrm{SW}}, T_1) + \mathsf{b}(T_1)$. Since no weight change is enacted before $\mathsf{d}(T_1)$, $\mathsf{d}(T_1) = \mathcal{D}(\mathcal{I}_{\mathrm{SW}}, T_1) = 3$. Thus, by rule I, T "leaves" at time $\mathcal{D}(\mathcal{I}_{\mathrm{SW}}, T_1) + \mathsf{b}(T_1) = 4$. Notice that the difference in rule I between cases (i) and (ii) is that, when a task increases its weight, the weight change is immediately enacted, whereas when a task decreases its weight, its weight change is not enacted until time $\mathcal{D}(\mathcal{I}_{\mathrm{SW}}, T_j) + \mathsf{b}(T_j)$. Thus, T's scheduling weight is redefined at different times.

Throughout this paper we use PD^2-OI (respectively, PD^2-LJ) to refer to reweighting via rules O and I (resp., the leave/join rules L and J) under PD^2. Since these rules change the ordering of a task in the priority queues that determine scheduling, the time complexity for reweighting one task is $O(logN)$, where N is the number of tasks in the system.

4 Scheduling Correctness and Drift Bounds

In the prior section, we used $\mathcal{I}_{\mathrm{SW}}$ to determine the release times of future subtasks. Notice that $\mathcal{I}_{\mathrm{SW}}$ and PD^2-OI treat halted subtasks differently. Specifically, $\mathcal{I}_{\mathrm{SW}}$ will "partially" allocate a halted subtask, whereas PD^2-OI will never schedule a halted subtask. Because of this difference, it is convenient, when proving correctness and drift bounds, to slightly alter $\mathcal{I}_{\mathrm{SW}}$ by eliminating halted subtasks. Therefore, we use $\mathcal{I}_{\mathrm{CSW}}$ instead, since $\mathcal{I}_{\mathrm{CSW}}$ does not allocate any capacity to any halted subtask. In this section, we discuss the scheduling correctness of PD^2-OI (the full proof can be found in an expanded version of this paper, which is available as a technical report [7]), formally define drift, and discuss the drift bounds of PD^2-LJ, PD^2-OI, and any EPDF reweighting algorithm.

We first show that initiating multiple reweighting events without enacting them does not increase the time of the next weight-change enactment. We show this by proving the following.

(C) If T initiates two weight-change events at t_c and t'_c, where $t_c < t'_c$ and $t'_c < t_e$, and t_e and t'_e denote the time the changes initiated at t_c and t'_c, respectively, would have been enacted in the absence of other reweighting events, then $t'_e \leq t_e$.

Proof of (C). Assume that t_c, t_e, t'_c, and t'_e are as defined in (C). Notice that any type of reweighting events initiated at t_c except for ideal-changeable decreasing-weight events and (some) omission-changeable events, are enacted within one quantum. Thus, we assume that T is either omission-changeable (and not immediately enacted) or decreasing-weight ideal-changeable at t_c. Before continuing, notice that if $\mathsf{d}(T_j) \leq t'_c$, where T_j is as defined in rules O and I, then the change initiated at t'_c is enacted by $t'_c + 1$. Since $t'_c < t_e$, this implies that $t'_e \leq t_e$. Thus, we assume in the rest of the proof that $t'_c < \mathsf{d}(T_j)$. We first consider the case wherein T is omission-changeable at t_c and this change is not immediately enacted. In this case, the change initiated at t_c is enacted at time $t_e = \mathcal{D}(\mathcal{I}_{\mathrm{SW}}, T_{j-1}) + \mathsf{b}(T_{j-1})$. Since T is omission-changeable at t_c, it is halted at t_c and no successor subtask can be released until the change initiated at t_c (or a future change) has been enacted. Hence, since $t'_c < t_e$, T_{j+1} is not released until after t'_c. Since $\mathsf{r}(T_j) < t_c < t'_c < \mathsf{d}(T_j)$, T_j is the last-released subtask of T at or before t'_c. Because T_j was halted at t_c, T is therefore omission-changeable at t'_c. Thus, by rule O, the change initiated at t'_c is enacted at time $t'_e = \mathsf{min}(t'_c, \mathcal{D}(\mathcal{I}_{\mathrm{SW}}, T_{j-1}) + \mathsf{b}(T_{j-1}))$. Thus, $t'_e = t_e$. We next consider the case wherein T is decreasing-weight ideal-changeable at t_c. By rule I, such a change initiated at t_c will be enacted at time $t_e = \mathcal{D}(\mathcal{I}_{\mathrm{SW}}, T_j) + \mathsf{b}(T_j)$. Since T is ideal-changeable at t_c, no subtask can be released until the change which was initiated at t_c (or a future change) has been enacted. Hence, since $t'_c < t_e$, T_{j+1} is not released until after the change at t'_c is initiated. Since $\mathsf{r}(T_j) < t_c < t'_c < \mathsf{d}(T_j)$, T_j is the last-released subtask of T at or before t'_c. Because T_j is scheduled before t'_c, T is ideal-changeable at t'_c. If the event at t'_c is a decreasing-weight event, then by rule I, it is enacted at time $t'_e = \mathcal{D}(\mathcal{I}_{\mathrm{SW}}, T_j) + \mathsf{b}(T_j) = t_e$; if it is an increasing-weight event, then $t'_e = t'_c < t_e$. □

When Srinivasan and Anderson [12] proved the scheduling correctness for PD^2-LJ for an IS task system, they assumed that the weight of all tasks is at most M and utilized the property that in an IS system, the windows for any subtask T_j and its successor T_{j+1} do not "overlap" by more than $\mathsf{b}(T_j)$ quanta, *i.e.*, $\mathsf{d}(T_j) - \mathsf{b}(T_j) \leq \mathsf{r}(T_{j+1})$. However, this property can be weakened without affecting most of their proof, so that their proof can be applied to an AIS task system. Specifically, their proof can be used to establish the scheduling correctness of PD^2-OI for any AIS task system τ, if the following conditions hold, which parallel the assumption that the weight of all tasks is at most M and the property that in an IS system $\mathsf{d}(T_j) - \mathsf{b}(T_j) \leq \mathsf{r}(T_{j+1})$. (In these properties, we denote the PD^2-OI schedule of τ as $\mathcal{S}$.)

(W) For any time t, $\sum_{T \in \tau} \mathsf{swt}(T, t) \leq M$, where M is the number of processors.

(V) For the subtasks T_i and T_{i+1}, if $\mathsf{d}(T_i) - \mathsf{b}(T_i) > \mathsf{r}(T_{i+1})$, then $\mathcal{D}(\mathcal{I}_{\mathrm{CSW}}, T_i) \leq \mathsf{r}(T_{i+1})$ and $\mathcal{D}(\mathcal{S}, T_i) \leq \mathsf{r}(T_{i+1})$.

Since (W) can be satisfied by policing weight-change requests, we focus our attention on showing that $\mathcal{S}$ and $\mathcal{I}_{\mathrm{CSW}}$ satisfy (V).

Proof of (V). Before we begin, notice that by the rules O and I, when T initiates a weight change at time t_c, it is enacted no later than time $\mathsf{r}(T_k)$, where T_k is the next-released subtask of T. (By (C), no sequence of reweighting events can delay the next weight-change enactment after a weight change has been initiated, and by the rules I and O, a subtask is released within one quantum of a weight-change enactment. Thus, T_k is eventually released.) Hence, if a weight change is enacted over the range $(\mathsf{r}(T_\ell), \mathsf{d}(T_\ell)]$, then that change must have been initiated over the range $[\mathsf{r}(T_\ell), \mathsf{d}(T_\ell)]$.

Let T_i be some subtask such that $\mathsf{d}(T_i) - \mathsf{b}(T_i) > \mathsf{r}(T_{i+1})$. By the definition of a subtask release, if $\mathsf{d}(T_i) - \mathsf{b}(T_i) > \mathsf{r}(T_{i+1})$, then T enacted a weight change at $t_e \in (\mathsf{r}(T_i), \mathsf{r}(T_{i+1})]$; otherwise, we would have $\mathsf{d}(T_i) - \mathsf{b}(T_i) \leq \mathsf{r}(T_{i+1})$, by Eqn. (4). Since a weight change is enacted in the range $(\mathsf{r}(T_i), \mathsf{d}(T_i)]$, (as established above) a change must have been initiated in the range $[\mathsf{r}(T_i), \mathsf{d}(T_i)]$. Thus, since a weight change is initiated before it is enacted (and by assumption $t_e \leq \mathsf{r}(T_{i+1}) < \mathsf{d}(T_i) - \mathsf{b}(T_i) \leq \mathsf{d}(T_i)$), a change must have been initiated in $[\mathsf{r}(T_i), \mathsf{r}(T_{i+1})]$. Without loss of generality, let t_c be the earliest time in this range that T initiates a weight change.

At time t_c, T is either omission- or ideal-changeable. We now consider these two cases. If at t_c, T is omission-changeable, then by rule O, T_i is halted at t_c. In this case, T_i is complete by $t_c \leq t_e \leq \mathsf{r}(T_{i+1})$ in both $\mathcal{S}$ and the $\mathcal{I}_{\mathrm{CSW}}$ schedule. If T is ideal-changeable at t_c, then T_i has been scheduled in $\mathcal{S}$ before t_c, and hence, T_i is complete by t_c in $\mathcal{S}$. Furthermore, in this case T_{i+1} is not released until time $\mathcal{D}(\mathcal{I}_{\mathrm{CSW}}, T_i) + \mathsf{b}(T_i)$. $\square$

By (V), it is possible to use Srinivasan and Anderson's correctness proof for PD^2-LJ [12] to prove the following theorem. (The full proof can be found in [7].)

Theorem 2. *Under PD^2-OI, no subtask is scheduled after its scheduling deadline, provided that (W) holds.*

4.1 Drift

We now turn our attention to the issue of measuring drift under PD^2-OI. For most real-time scheduling algorithms, the difference between the ideal and actual allocations a task receives lies within some bounded range centered at zero (that is, *lag bounds* are maintained). For example, under PD^2 (*i.e.*, PD^2-OI without weight changes), the difference between the ideal and actual allocations for a task lies within $(-1, 1)$. When a weight change occurs, the same range is maintained except that it may be centered at a different value. This lost allocation is called *drift*. In general, a task's drift per reweighting event will be non-negative (non-positive) if it increases (decreases) its weight. Let $\mathcal{S}$ denote a PD^2-OI schedule of some task system τ. Since Thm. 2 established that no subtask misses its deadline, and neither $\mathcal{S}$ nor $\mathcal{I}_{\mathrm{CSW}}$ schedules any halted subtasks, $\mathsf{A}(\mathcal{S}, T, 0, t)$ differs from $\mathsf{A}(\mathcal{I}_{\mathrm{CSW}}, T, 0, t)$ by ± 1. Hence, we can bound the drift that a task T incurs under PD^2-OI up to time t by comparing T's total allocation up to time t in $\mathcal{I}_{\mathrm{CSW}}$ to that in the "ideal processor sharing" schedule, in which all weight changes are enacted instantaneously.

In the *ideal processor sharing* ($\mathcal{I}_{\mathrm{PS}}$) schedule, at each instant t, each task T in τ is allocated a share equal to its weight $\mathsf{wt}(T, t)$. Hence, over the interval $[t_1, t_2)$, the task T is allocated $\mathsf{A}(\mathcal{I}_{\mathrm{PS}}, T, t_1, t_2) = \int_{t_1}^{t_2} \mathsf{wt}(T, u)du$ time. (For the remainder of this section, we assume that every subtask in T is released as early as possible. This assumption can be removed at the cost of more complex notation. If we did not make this assumption, then the allocation function for $\mathcal{I}_{\mathrm{PS}}$ would equal zero between active subtasks.) $\mathcal{I}_{\mathrm{PS}}$ is similar to $\mathcal{I}_{\mathrm{SW}}$ and $\mathcal{I}_{\mathrm{CSW}}$, with three major exceptions: **(i)** tasks in $\mathcal{I}_{\mathrm{PS}}$ continually receive allocations, whereas tasks in $\mathcal{I}_{\mathrm{SW}}$ and $\mathcal{I}_{\mathrm{CSW}}$ receive allocations only at quantum boundaries; **(ii)** under $\mathcal{I}_{\mathrm{PS}}$, each task receives an allocation equal to its *weight*, whereas under $\mathcal{I}_{\mathrm{SW}}$ and $\mathcal{I}_{\mathrm{CSW}}$, each task receives allocations according to its *scheduling weight*; and **(iii)** the total allocation each task receives in $\mathcal{I}_{\mathrm{SW}}$ and $\mathcal{I}_{\mathrm{CSW}}$ is calculated based on the releases and completion times of its active subtasks, whereas allocations in $\mathcal{I}_{\mathrm{PS}}$ are independent of subtask releases and completion times. Hence, even if all active subtasks of a given task are halted, $\mathcal{I}_{\text{PS}}$ still allocates capacity to that task. For example, consider Fig. 7, which depicts the allocations in the schedules $\mathcal{I}_{\text{CSW}}$ and $\mathcal{I}_{\text{PS}}$ (insets (a) and (b), resp.) to a task X that has an initial weight of $3/19$ that increases to $2/5$ (via rule I) at time 8. Notice that in $\mathcal{I}_{\text{PS}}$ over the range $[9, 11)$, X receives an allocation equal to its weight at every instant (for a total allocation of $4/5$ over $[9, 11)$). Compare this to $\mathcal{I}_{\text{CSW}}$, in which X receives only an allocation of $32/95$ over the same range.

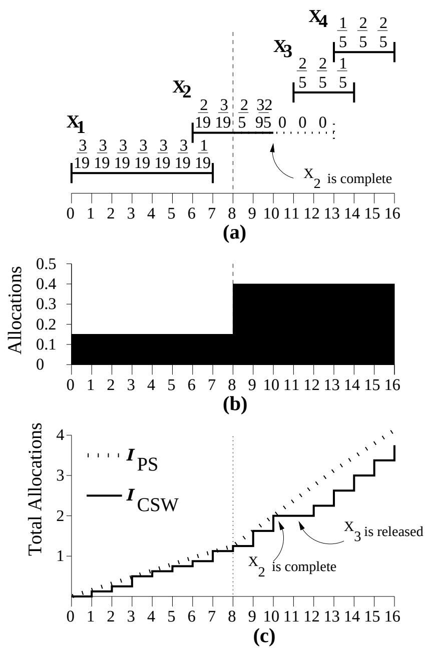


Figure 7: Allocations for a task X with an initial weight of $3/19$ that changes to $2/5$. **(a)** The value of $\mathsf{A}(\mathcal{I}_{\text{CSW}}, X_j, u)$ for each slot and subtask. **(b)** The allocations to X in $\mathcal{I}_{\text{PS}}$ at each instant. **(c)** The total allocations to X in $\mathcal{I}_{\text{CSW}}$ and $\mathcal{I}_{\text{PS}}$.

Formally, under PD^2-OI, the *drift of a task T is defined*[1] as

$$\mathsf{drift}(T, t) = \mathsf{A}(\mathcal{I}_{\text{PS}}, T, 0, u) - \mathsf{A}(\mathcal{I}_{\text{CSW}}, T, 0, u), \tag{5}$$

where u is defined as follows: if $t < \mathsf{r}(T_1)$, then $u = t$; otherwise, $u = \mathsf{r}(T_i)$, where T_i is the last-released subtask of T at or before t such that $\mathsf{Id}(T_i) = i$. For example, in Fig. 6(b), the drift of task T at time $t = 9$ is $\mathsf{A}(\mathcal{I}_{\text{PS}}, T, 0, \mathsf{r}(T_1)) - \mathsf{A}(\mathcal{I}_{\text{CSW}}, T, 0, \mathsf{r}(T_1)) = 0 - 0 = 0$, whereas at time $t = 10$, the drift of T is $\mathsf{A}(\mathcal{I}_{\text{PS}}, T, 0, \mathsf{r}(T_3)) - \mathsf{A}(\mathcal{I}_{\text{CSW}}, T, 0, \mathsf{r}(T_3)) = 3/2 - 1 = 1/2$. Notice that, since T_2 is halted at time 10, $\mathsf{A}(\mathcal{I}_{\text{CSW}}, T_2, 0, 10) = 0$. We say that a reweighting algorithm is *fine-grained* iff there exists some constant value c such that the drift per weight change is less than c. We say that a reweighting algorithm is *coarse-grained* otherwise.

We now prove that PD^2-LJ is not fine-grained.[2] Consider the four-processor system depicted in Fig. 8. This 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. By rule L, T cannot "leave" until time 10. Hence, the change is not enacted until time 10. Thus, over the range $[4, 10)$, T receives a $1/10$ per-slot allocation in $\mathcal{I}_{\text{CSW}}$ and $1/2$ in $\mathcal{I}_{\text{PS}}$. Hence, T's drift reaches a value of $24/10$ at time 10. This example can be generalized to generate any value of drift for T, by decreasing its initial weight. Under PD^2-LJ, 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 $\mathsf{drift}(T, \mathsf{d}(T_1)) \geq c$. The theorem below follows.

Theorem 3. PD^2-LJ *is not fine-grained.*

I_{PS}	0	$\frac{1}{10}$	$\frac{2}{10}$	$\frac{3}{10}$	$\frac{4}{10}$	$\frac{9}{10}$	$\frac{14}{10}$	$\frac{19}{10}$	$\frac{24}{10}$	$\frac{29}{10}$	$\frac{34}{10}$	$\frac{39}{10}$	$\frac{44}{10}$	$\frac{49}{10}$	$\frac{54}{10}$
I_{CSW}	0	$\frac{1}{10}$	$\frac{2}{10}$	$\frac{3}{10}$	$\frac{4}{10}$	$\frac{5}{10}$	$\frac{6}{10}$	$\frac{7}{10}$	$\frac{8}{10}$	$\frac{9}{10}$	$\frac{10}{10}$	$\frac{15}{10}$	$\frac{20}{10}$	$\frac{25}{10}$	$\frac{30}{10}$
drift(T,t)	0	0	0	0	0	0	0	0	0	0	$\frac{24}{10}$	$\frac{24}{10}$	$\frac{24}{10}$	$\frac{24}{10}$	$\frac{24}{10}$

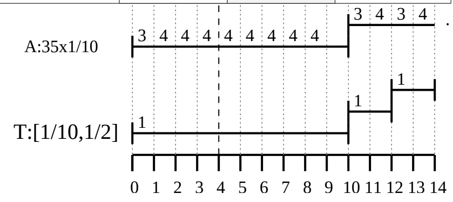


Figure 8: The PD^2-LJ schedule of a four-processor system with a set A of 35 tasks with weight 1/10 and a task T with weight 1/10 that increases to 1/2 at time 4. The allocations of T in $\mathcal{I}_{\text{CSW}}$ and $\mathcal{I}_{\text{PS}}$ and its drift are labeled.

Next, we show that any EPDF scheduling algorithm incurs some drift.[3] This follows from the two-processor counterexample depicted in Fig. 9. This system consists of a set A of 10 tasks with weight 1/7

[1]The definition of drift presented in (5) is designed specifically for EPDF systems. This concept can be more generally defined to pertain to other systems like global EDF and partitioning schemes.

[2]Given that PD^2-OI is simply an extension of PD^2-LJ that allows faster reweighting, it should be clear that the PD^2-OI-centric definition of drift given in (5) can be extended to apply to PD^2-LJ by simply basing the definition of $\mathcal{I}_{\text{CSW}}$ on PD^2-LJ instead of PD^2-OI.

[3]As before, the drift definition in (5) can be applied to such an algorithm by simply redefining $\mathcal{I}_{\text{CSW}}$ accordingly.

that leave at time 7, a set B of two tasks with weight 1/6 that leave at time 6, a set C of two tasks with weight 1/14 that join at time 6, and a set D of five tasks with a weight of 1/21 that increases to 1/3 at time 7. With subtask deadlines defined by $\mathcal{I}_{\mathrm{PS}}$, the deadline for each task in set D changes at time 7 from 21 to 9. The tasks in D have an original deadline of 21 because that is the projected time at which their $\mathcal{I}_{\mathrm{PS}}$ allocations will equal one if their weights do not change. These tasks change their deadlines to 9 at time 7 because the new weight, 1/3, changes the projected time by which their $\mathcal{I}_{\mathrm{PS}}$ allocations will equal one to time 9. Hence, any EPDF algorithm will not schedule the tasks in D until time 7. As a result, a deadline is missed. Notice that any EPDF algorithm would need to use projections for determining subtask deadlines if we assume no prior knowledge of weight changes. To prevent a deadline miss, the lag-bound range must be shifted, thus incurring drift. The theorem below follows.

Theorem 4. *All* EPDF *algorithms can incur non-zero drift per reweighting event.*

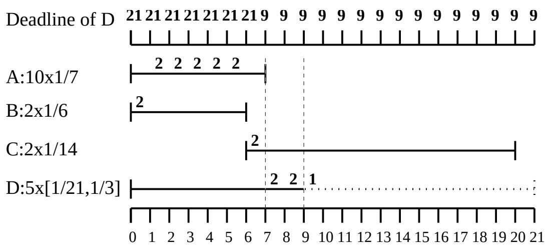


Figure 9: A two-processor system consisting of a set A of 10 tasks with weight 1/7 that leave at time 7, a set B of two tasks with weight 1/6 that leave at time 6, a set C of two tasks with weight 1/14 that join at time 6, and a set D of five tasks with a weight of 1/21 that increases to 1/3 at time 7. The projected deadlines of tasks in D based on their true ideal allocations are labeled above. Notice that a task in D misses its deadline at time 9 since its "true" deadline is unknown until time 7.

Finally, we show that PD^2-OI is fine grained. By the definition of drift, in order to prove that PD^2-OI is fine-grained, we merely need to consider the window placement of a task after it is reweighted. Suppose that a task T initiates a weight change at t_c. Let t_e be the next time at which T enacts a change at or after t_c, and let T_j be the last-released subtask of T at or before t_c, if T_j exists. (If T_j does not exist, then T's drift does not change.) We now show that if the change initiated at t_c is enacted at t_e, then the added drift is bounded by showing that the maximal absolute difference between $\mathcal{I}_{\mathrm{CSW}}$ and $\mathcal{I}_{\mathrm{PS}}$ in allocations to T over the interval $[t_c, \mathsf{r}(T_{j+1}))$ is at most two. Notice that such a result implies that if T were to initiate a change at time t'_c such that $t_c < t'_c \leq t_e$, then the absolute difference in allocation between $\mathcal{I}_{\mathrm{CSW}}$ and $\mathcal{I}_{\mathrm{PS}}$ to T over the interval $[t_c, t'_c)$ is at most two. This implies that the absolute value of the added drift per reweighting event is at most two, even for those events that are "canceled" by future reweighting events that occur before any change is enacted.

If $\mathsf{d}(T_j) \leq t_c$, then the weight change is enacted within one quantum. Since the maximal weight of a task is $1/2$, the maximal increase in the absolute value of drift in such a case is $1/2$. If T_j exists and $\mathsf{r}(T_j) \leq t_c < \mathsf{d}(T_j)$, then T is either omission- or flow-changeable. If T changes its weight via rule O, then the resulting allocation error is at most two quanta. One quantum of the error can be incurred because T_j is halted at t_c, resulting in an allocation of up to one subtask being "lost." For example, in Fig. 6(b), $\mathsf{A}(\mathcal{I}_{\mathrm{CSW}}, T_2, 0, 10) = 1/2$ quanta is "lost" when T initiates and enacts a weight change at time 10. The second quantum of error can be incurred because the change T initiated at t_c may not be enacted until time $\mathsf{max}(t_c, \mathcal{D}(\mathcal{I}_{\mathrm{CSW}}, T_{j-1}) + \mathsf{b}(T_{j-1}))$. By (V), if a change is enacted in the range $[\mathsf{r}(T_{j-1}), \mathsf{d}(T_{j-1})]$, then $\mathcal{D}(\mathcal{I}_{\mathrm{CSW}}, T_{j-1}) - \mathsf{r}(T_j) \leq \mathsf{b}(T_{j-1})$. Further, by Eqn. (4), if no change is enacted in the range $[\mathsf{r}(T_{j-1}), \mathsf{d}(T_{j-1})]$, then $\mathcal{D}(\mathcal{I}_{\mathrm{CSW}}, T_{j-1}) - \mathsf{r}(T_j) \leq \mathsf{b}(T_{j-1})$. Thus, since $\mathsf{r}(T_j) \leq t_c$, the range $[t_c, \mathsf{max}(t_c, \mathcal{D}(\mathcal{I}_{\mathrm{CSW}}, T_{j-1}) + \mathsf{b}(T_{j-1})))$ has a length of at most two. Hence, the change initiated at t_c may not be enacted for two quanta, and since the maximal weight for any task is $1/2$, the $\mathcal{I}_{\mathrm{PS}}$ allocations may "get ahead" (if T increases its weight at

t_c) or "fall behind" (if T decreases its weight at t_c) by $2 \cdot 1/2 = 1$ quantum.

If T increases its weight at t_c via rule I, then the weight change is immediately enacted and T_{j+1} is released at time $\mathcal{D}(\mathcal{I}_{\text{CSW}}, T_j) + \mathsf{b}(T_j)$. Since the weight change is immediately enacted, the only period of time during which T receives less allocation in $\mathcal{I}_{\text{CSW}}$ than in $\mathcal{I}_{\text{PS}}$ is between $\mathcal{D}(\mathcal{I}_{\text{CSW}}, T_j) - 1$ and $\mathsf{r}(T_{j+1})$. Since $\mathsf{r}(T_{j+1}) = \mathcal{D}(\mathcal{I}_{\text{CSW}}, T_j) + \mathsf{b}(T_j)$, the length of this interval is at most two. Since the maximal weight of a task is $1/2$, the maximal increase in drift is $2 \cdot 1/2$. For example, in Fig. 6(c), $\mathsf{A}(\mathcal{I}_{\text{CSW}}, T, 10, 12) = \mathsf{A}(\mathcal{I}_{\text{CSW}}, T, 0, 12) - \mathsf{A}(\mathcal{I}_{\text{CSW}}, T, 0, 10) = 4/2 - 3/2 = 1/2 < 1 = 2 - 1 = \mathsf{A}(\mathcal{I}_{\text{PS}}, T, 0, 12) - \mathsf{A}(\mathcal{I}_{\text{PS}}, T, 0, 10) = \mathsf{A}(\mathcal{I}_{\text{PS}}, T, 10, 12)$. If T decreases its weight at t_c via rule I, then T_{j+1} is released at time $\mathcal{D}(\mathcal{I}_{\text{CSW}}, T_j) + \mathsf{b}(T_j)$. Since T decreases its weight, over the range $[t_c, \mathcal{D}(\mathcal{I}_{\text{CSW}}, T_j))$, T is allocated at most one quantum more in $\mathcal{I}_{\text{CSW}}$ than in $\mathcal{I}_{\text{PS}}$. Furthermore, over the range $[\mathcal{D}(\mathcal{I}_{\text{CSW}}, T_j), \mathcal{D}(\mathcal{I}_{\text{CSW}}, T_j) + \mathsf{b}(T_j))$, T is allocated at most $1/2$ quanta more in $\mathcal{I}_{\text{PS}}$ than in $\mathcal{I}_{\text{CSW}}$, since the length of this range is one and the maximal weight of a task is $1/2$. Thus, the maximal possible decrease in drift is one and the maximal possible increase in drift is $1/2$. For example, in Fig. 6(d), the drift incurred by changing the weight of T from $2/5$ to $3/20$ is $-3/20$, *i.e.*, $\mathsf{drift}(T, t) = -3/20$, where $t \geq 4$.

Theorem 5. PD^2-OI *is fine-grained; moreover, the absolute value of the per-event drift under* PD^2-OI *for each task is at most two.*

5 Experimental Results

The results of this paper are part of a longer-term project on adaptive real-time allocation in which Whisper, described earlier, will be used as a test application. In this section, we provide an extensive simulation of Whisper. Unfortunately, at this point in time, it is not feasible to produce experiments involving a real implementation of Whisper for several reasons. First, the existing Whisper system is single threaded (and non-adaptive) and consists of several thousand lines of code. All of this code has to be re-implemented as a multi-threaded system, which is a nontrivial task. Indeed, because of this, it is *essential* that we first understand the algorithmic tradeoffs involved in adapting tasks on multiprocessor real-time systems. The purpose of this paper, as well as the two related ones we have written on adaption under partitioning [4] and global EDF [8], is to explore these tradeoffs. Additionally, this paper is only concerned with scheduling methods that facilitate adaption—we have *not* addressed the issue of devising mechanisms for determining *how* and *when* the system should adapt. Such mechanisms will be based on issues involving virtual-reality systems that are well beyond the scope of this paper. For these reasons, we have chose to evaluate PD^2-OI via simulation.

Specifically, we present a simulated implementation of Whisper on a four-processor system, with 2.7 GHz processors and a 1-ms quantum. The system was simulated for 10 s, with a sampling frequency of 1,000 Hz for each tracked object. Whisper tracks users through a system of speakers attached to users and microphones attached to the ceiling. Each speaker emits a unique "white noise" signal that is received by the microphones. As depicted in Fig. 10, we simulated three speakers (one per object) revolving around a 5-cm pole in a 1m $\times$ 1m room with a microphone in each corner. The pole creates potential occlusions. Whisper is able to compute the time-shift between the transmitted and received versions of the sound by performing a *correlation* calculation on the most recent set of samples. As the distance between the speaker and microphone changes, so does the number of correlation computations necessary to correctly track the speaker. This distance is (obviously) impacted by a speaker's movement, but is also lengthened when an occlusion is caused by the pole. The range of weights of each task was determined (as a function of a tracked object's position) by implementing and timing the basic computation of the correlation algorithm (an accumulate-and-multiply operation) on a testbed system that is the same as that assumed in the simulations.

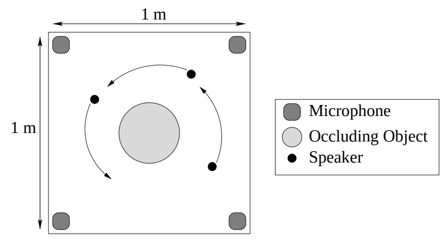


Figure 10: The simulated Whisper system.

Each simulation was run 61 times with the speakers placed randomly around the pole, at an equal distance from the pole, and each rotating around the pole at the same speed. As mentioned above, as the distance between a speaker and microphone changes, so does the amount of computation necessary to correctly track the speaker. This distance is (obviously) impacted by a speaker's movement, but is also lengthened by an occlusion.

In our simulations, we made several simplifying assumptions. First, all objects are moving in only two dimensions. Second, there is no ambient noise in the room. Third, no speaker can interfere with any other speaker. Fourth, all objects move at a constant rate. Fifth, for each speaker/microphone pair, there is only one task. Sixth, the weight of each task changes only once for every 5 cm of distance between its associated speaker and microphone. Finally, all speakers and microphones are omnidirectional. Because there are three speakers in this simulation, there is not sufficient capacity on the assumed system to statically allocate each task the capacity it needs to perform all calculations in the worst case. Even with theses assumptions, frequent share adaptations are required, since the share required by each task changes with the distance between its associated speaker/microphone pair. (In the absence of these assumptions, we expect PD^2-LJ to be completely inadequate, since required adaptations would be even more pronounced and frequent than those occurring here.)

We conducted experiments in which we varied the distance of each object from the center of the room from 10 cm to 50 cm, the speed of each object from 0.1 m/s to 3.5 m/s (such speeds typify human motion), and the presence of an occluding object (the pole). However, due to page limitations, the graphs below present only a representative sampling of the data we collected. All simulations are run for 1,000 time steps (10 s assuming a 1 ms quantum). While the ultimate metric for determining the efficacy of a tracking system would be user perception, this metric is not currently available for reasons discussed earlier. Thus, we compared PD^2-OI and PD^2-LJ by measuring the deviance of each from $\mathcal{I}_{\mathrm{PS}}$. This metric should provide us with a reasonable impression of how well these systems will fare when Whisper is fully re-implemented. We implemented and timed both reweighting algorithms considered in our simulations on an actual testbed that is the same as that assumed in our simulations, and found that all per-slot scheduling decisions could be made in approximately 5 μs for all task systems in our experiments. We considered this value to be negligible in comparison to a 1-ms quantum and thus did not consider scheduling overheads in our simulations. In each graph presented below, 98% confidence intervals are given.

In the first two graphs, in Fig. 11(a) and (b), the distance from the center of the room to each speaker is 25 cm, and the speed at which the speakers move varies from 0.5 m/s to 3.5 m/s. Inset (a) depicts the maximal drift of any task in the system at time 1,000 as a function of the speed of the speakers. Inset (b) gives the per-task average total amount of computation completed by

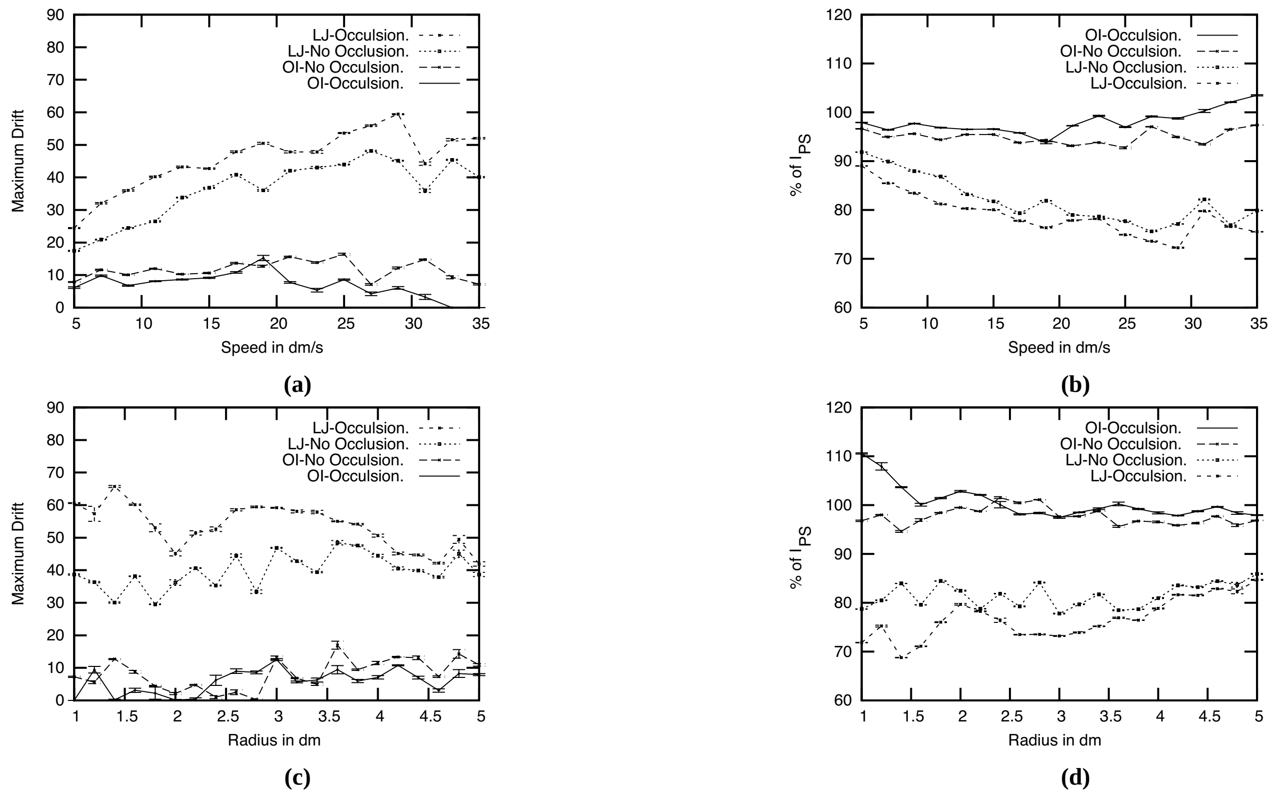


Figure 11: Selection of experiments. For clarity, the legend in each inset orders the curves in the (top-to-bottom) order they appear in that graph. **(a)** Maximum drift as a function of object speed. **(b)** Percent of ideal allocation (as defined by $\mathcal{I}_{\text{PS}}$) as a function of object speed. **(c)** Maximum drift as a function of radius of rotation. **(d)** Percent of ideal allocation as a function of radius of rotation.

time 1,000, as a percentage of the task's allocations in $\mathcal{I}_{\text{PS}}$, as a function of the speed of the speakers. Notice that PD^2-LJ's performance decreases (*i.e.*, maximal drift increases and the percentage of the $\mathcal{I}_{\text{PS}}$ allocation decreases) with an increase in speed. PD^2-OI's performance, on the other hand, improves with speed. The most probable explanation for this is that not all weight changes incur the same amount of drift. In particular, ideal-changeable tasks (*i.e.*, tasks that are reweighted after being scheduled) incur little drift under PD^2-OI. However, when a ideal-changeable task's weight change is "enacted" by PD^2-LJ, the amount of drift can be substantial. Hence, it is likely that, as the speed of the speakers increases, the number of ideal-changeable tasks also increases. (Note that the system is not fully loaded, so a task can receive more than 100% of its ideal allocation.)

In the second two graphs, in Fig. 11(c) and (d), the speed of the speakers is 2.9 m/s, and the distance from the center of the room to each speaker varies from 10 cm to 50 cm. Inset (c) depicts the maximal drift of any task in the system at time 1,000 as a function of the distance of the speakers. Inset (d) gives the per-task average total amount of computation completed by time 1,000, as a percentage of the task's allocations in $\mathcal{I}_{\text{PS}}$, as a function of the distance of the speakers. One interesting behavior in inset (c) is that the performance of PD^2-LJ, in the presence of occlusions, improves as the distance increases. Such behavior is likely a consequence of the fact that, as the radius of the speakers increases and the speed remains constant, the distance between each speaker and microphone is affected by the occluding object for longer periods of time. Hence, share changes that occur as a speaker becomes (or ceases being) occluded are less frequent, thus improving performance.

Note that the Whisper experiments presented here are fairly generous to PD^2-LJ. While weight changes occur frequently, all weight changes are of one order of magnitude; in fact, most weight changes are fairly incremental. However, even in this scenario, PD^2-LJ completes at most 85% of the allocations in $\mathcal{I}_{\rm PS}$, while PD^2-OI *is always* within 95% of $\mathcal{I}_{\rm PS}$.

6 Concluding Remarks

We have shown (for the first time) that fine-grained reweighting is possible on fair-scheduled multiprocessor platforms. The experiments reported herein show that our reweighting rules enable greater precision in adapting than PD^2-LJ. 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, PD^2-LJ entails only $O(M \log N)$ time. However, as 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" PD^2-OI and PD^2-LJ [5].

As mentioned earlier in this paper, we have ignored the issue of reweighting heavy tasks. The inclusion of heavy tasks complicates the reweighting rules, since such a task can release a new subtask with a window length of two in (nearly) every time slot. As a result, one "wrong" scheduling decision can force a cascade of "wrong" scheduling decisions. For non-adaptive systems, it is possible to calculate the length of such a cascade, and make scheduling decisions based on that information. However, for adaptive systems, the lengths of these cascades will change with time. Thus, when scheduling adaptive heavy tasks, particular care must be taken to "correct" such "cascades." Because of the complexity involved in constructing and proving reweighting rules for heavy tasks, we refer the reader to the first author's upcoming Ph.D. dissertation, which addresses this issue.

One major drawback to PD^2-OI scheduling is that it (like all Pfair algorithms) suffers from potentially high migration and preemption costs. These costs can be mitigated by using adaption schemes based upon partitioning [4] and global EDF [8]. However, as noted earlier, under partitioning, fine-grained reweighting is (provably) impossible; and under global EDF, it is possible only if deadline misses are permissible. Because of these various tradeoffs, all three approaches are of value.

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 [9]). Both 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 *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 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 *Proceedings of the 9th International Parallel Processing Symposium*, pages 280–288, 1995.

[4] A. Block and J. Anderson. Accuracy versus Migration Overhead in Multiprocessor Reweighting Algorithms. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems*, pages 355–364. IEEE, July 2006.

[5] A. Block and J. Anderson. Task reweighting on multiprocessors: Efficiency versus accuracy. In *Proceedings of the 13th International Workshop on Parallel and Distributed Real-Time Systems*, 2005, on CD ROM.

[6] A. Block, J. Anderson, and G. Bishop. Fine-grained task reweighting on multiprocessors. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 329–435. IEEE, August 2005.

[7] A. Block, J. Anderson, and G. Bishop. Fine-grained task reweighting on multiprocessors. Technical Report TR06-008, Department of Computer Science, The University of North Carolina at Chapel Hill, April 2006.

[8] A. Block, J. Anderson and U. Devi. Task Reweighting under Global Scheduling on Multiprocessors. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 128–138. IEEE, July 2006.

[9] C. Lu, J. Stankovic, G. Tao, and S. Son. Design and evaluation of a feedback control edf scheduling algorithm. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 44–53, 1999.

[10] 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.

[11] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. *Journal of Computer and System Sciences*, 72(6):1094–1117, 2006.

[12] A. Srinivasan and J. Anderson. Fair scheduling of dynamic task sys. on multiprocessors. In *Proceedings of the 11th International Workshop on Parallel and Distributed Real-Time Systems*, 2003, on CD ROM.

[13] 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 *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 288–299, 1996.

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