Accuracy versus Migration Overhead in Multiprocessor Reweighting Algorithms*

Aaron Block and James H. Anderson
Department of Computer Science
University of North Carolina at Chapel Hill

October 2005

Abstract

We consider schemes for enacting task share changes—a process called *reweighting*—on real-time multiprocessor platforms. Our particular focus is reweighting schemes that are deployed in environments in which tasks may *frequently* request *significant* share changes. Prior work has shown that fair scheduling algorithms are capable of reweighting tasks with minimal allocation error; this source of error is defined by comparing to an ideal allocation scheme. However, such algorithms do so at the expense of potentially high task-migration overheads. While in theoretical research it is common to ignore migration overheads, they actually constitute an additional source of error. With frequent and significant share changes, task migrations cannot be entirely prevented, if reasonable allocation error is desired. However, partitioning-based schemes that allow *occasional* reassignments of tasks to processors have the potential of significantly reducing migration costs. On the other hand, such schemes cannot match fair schemes with respect to allocation error, because under partitioning, some share allocations may not be possible. In this paper, we consider the question of whether the lower migration costs of partitioning-based schemes are sufficient to compensate for their greater allocation error. We show that allocation error in such schemes is influenced by several factors. We suggest several approaches for dealing with these factors and compare one of the resulting schemes to a prior fair scheme. Our conclusion is that partitioning-based schemes are capable of providing significantly lower overall error (due to *both* allocation inaccuracies *and* migration costs) than fair schemes in the *average* case. However, partitioning-based schemes are incapable of providing comparable fairness and real-time guarantees.

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

1 Introduction

Real-time systems that are *adaptive* in nature have received considerable recent attention [3, 7, 8]. In addition, *multiprocessor* platforms are of growing importance, due to both hardware trends such as the emergence of multicore technologies, and also to the prevalence of computationally-intensive applications for which single-processor designs are not sufficient. In a prior paper [3], we considered the use of fair scheduling algorithms to schedule highly-adaptive workloads on (tightly-coupled) multiprocessor platforms. Such workloads are characterized by the need to change the processor shares of tasks frequently and to a significant extent. Fair scheduling techniques have the advantage of ensuring good accuracy in enacting share changes, but do so at the expense of potentially frequent task migrations among processors. Thus, other scheduling approaches that may be less accurate, but migrate tasks less frequently, may still be of interest. In this paper, we consider the use of such approaches and consider the tradeoff between accuracy and migration costs in detail. Our specific focus is partitioning approaches that forbid task migrations (in the absence of share changes). This focus is justified by the wide-spread use of such approaches on multiprocessor platforms. The key issue we seek to address is whether the lower migration overheads in less migration-prone schemes is sufficient to compensate for their lower accuracy.

Whisper. To motivate the need for this work, we consider two example applications. The first of these is the Whisper tracking system, which was designed at the University of North Carolina to perform full-body tracking in virtual environments [9]. Whisper tracks users via an array of wall- and ceiling-mounted microphones that detect white noise emitted from speakers attached to each user's hands, feet, and head. Like many tracking systems, Whisper uses *predictive techniques* to track objects. The workload on Whisper is intensive enough to necessitate a multiprocessor design. Furthermore, adaptation is required because 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 will vary with time. In fact, the variance can be as much as *two orders of magnitude*. Moreover, adaptations must be enacted within *time scales as short as 10 ms.*

ASTA. Another application with similar requirements under development at the University of North Carolina, is the DARPA-funded ASTA video-enhancement system [2]. ASTA is capable of improving the quality of an underexposed video feed so that objects that are indistinguishable from the background become clear and in full color. In ASTA, darker objects require more computation to correct. Thus, as dark objects move in the video, the processor shares of the tasks assigned to process different areas of the video will change. ASTA will eventually be deployed in a military-grade full-color night vision system, so tasks will need to change shares as fast as a soldier's head can turn. In the planned configuration, a 10-processor multicore platform will be used.

Scheme	Drift	Overload	Migrations
PAS	*REQ*	W	only at weight-change events
PD^2-OF	2	0	every quantum

Table 1: Summary of worst-case results. *REQ* is the maximum request size. W is the weight of the $\left(M \cdot \lfloor \frac{1}{X} \rfloor + 1\right)^{st}$ "heaviest" task in the system, where M is the number of processors and X is the weight of the heaviest task. The PAS entries are tight in the sense that in any partitioned scheme, there exists a system that can cause a processor to be over-utilized by W, and in any EEVDF-based algorithm where deadlines can be missed by at most one quantum (like in EEVDF), drift can be as high as *REQ*.

Summary of results. While the terms "share," "weight," and "utilization" are often used interchangeably, we use *weight* to denote a task's desired utilization, and *share* to denote its actual guaranteed utilization. In each scheduling scheme we consider, a task's share is determined by its weight; in some of these schemes, the two are always equal, while in others, they may differ.[1] We refer to the process of enacting task weight/share changes as *reweighting*. Two reweighting-capable scheduling algorithms are considered: a previous fair algorithm developed by us called called PD^2-OF [3], which is a derivative of the PD^2 Pfair algorithm [1]; and a new algorithm called *partitioned-adaptive scheduling* (PAS), which is a derivative of Stoica *et al.*'s earliest-eligible-virtual-deadline-first (EEVDF) algorithm [8]. PAS is proposed herein as a good candidate partitioned algorithm.

Our results are summarized in Table 1, which lists the accuracy and migration cost of both schemes. Accuracy is assessed per reweighting event in terms of two quantities, "drift" and "overload error," which are measured in terms of the system's scheduling quantum size. *Drift* is the error, in comparison to an ideal allocation, that results due to a reweighting event [3]. (Under an ideal allocation, tasks are reweighted instantaneously, which is not possible in practice.) *Overload error* is the error that results from a scheduler's inability to give a task a share equal to its desired weight. This may happen under partitioning due to processor overloads. For example, it is impossible to assign a share of 2/3 to each of three tasks executing on two processors. One possibility is to assign two of the tasks to the same processor, giving each a share of 1/2. In this case, the difference between the weight and share of these tasks would be $2/3 - 1/2 = 1/6$. (The method by which we "distribute" any overload among tasks is a non-trivial issue, which we discuss in detail in Sec. 2.) Note that overload error is potentially more detrimental than drift: while drift is a one-time error assessed per reweighting event, overload error accumulates over time. As the example above suggests, under partitioned schemes, we cannot guarantee nonzero overload error, because of inevitable connections to bin-packing that arise. Another consequence of these connections is that, even under partitioning, migrations can happen. This is because some reweighting events may necessitate reassigning tasks to processors.

[1] In the proportional-share algorithm [8] that is basis of the new scheme we propose, weights are allowed to be arbitrary rational values. For consistency, we will always require them to range over $[0, 1]$.

In Table 1, *REQ* denotes the maximum amount of computation requested at one time by any task, and W denotes the desired processor share of the $(M \cdot \lfloor 1/X \rfloor + 1)^{st}$ "heaviest" task (by weight), where M is the number of processors and X is the weight of the heaviest task. Table 1 shows that algorithms that allow more frequent migrations, like the Pfair-based PD^2-OF algorithm, produce little drift and no overload error, and algorithms that restrict the frequency of migrations can produce substantial amounts of drift and overload error.

Pfair-based algorithms achieve the above properties by breaking the workload to be scheduled into quantum-length segments called *subtasks*, each of which is assigned a release time and a deadline. That is, the granularity of scheduling is much finer than is the case when arbitrary jobs are to be scheduled. Subtasks are scheduled on an earliest-deadline-first basis. Furthermore, different subtasks of a task may execute on different processors, *i.e.*, task migration is allowed. Accurate reweighting is possible within such a scheduling scheme because weight changes can be enacted in a fine-grained way by changing future subtask releases and deadlines. (It is worth pointing out that various subtleties arise in devising correct reweighting rules—see [3].)

Contributions. Our theoretical contributions include devising the PAS algorithm and associated reweighting rules, and establishing the error bounds for PAS in Table 1. The question that then remains is: for PAS and PD^2-OF, how do drift and overload error compare to any error due to migration costs? We attempt to answer this question via extensive simulation studies of Whisper and ASTA. In these studies, real migration costs were assumed based on actual measured values. These studies confirm the expectation that, if migration costs are high, then PAS performs well in the average case; however, PD^2-OF provides stronger real-time and fairness guarantees. Given our belief that PAS is a good candidate partitioned scheme, we conclude from this that, in applications where migration costs are low or high allocation accuracy is required, Pfair-based schemes are superior to other less migration-prone approaches. (As explained later, Whisper is such an application.) However, when average-case performance is more important or migration costs are high, a partitioned scheme may be the best choice. (As we explain later, ASTA is such an application.)

The rest of this paper is organized as follows. We begin in Sec. 2 by discussing the PAS algorithm in greater detail, and by establishing the properties mentioned above. Our experimental evaluation is then presented in Sec. 3. We conclude in Sec. 4.

2 Partitioning-Based Reweighting

In this section, we more thoroughly examine the issue of reweighting in partitioned systems. Because there cannot exist an optimal scheduling algorithm under partitioning, we focus our attention on different heuristic tradeoffs that can minimize different sources of error. Before we discuss these tradeoffs in detail, we first consider a fundamental limitation of all partitioning algorithms.

2.1 Limitations of Partitioning Schemes

Under any partitioning scheme, there exist feasible task systems that are not schedulable, even in the absence of weight changes. A commonly-cited example of this, mentioned earlier, is a two-processor system with three identical periodic tasks with an execution cost of 2.0 and a period of 3.0. Two of the initial jobs must execute on the same processor, thus over-utilizing it. There are two approaches for handling this problem. First, we could cap the total utilization of all tasks in the system. Unfortunately, under any M-processor partitioning scheme, a cap of approximately $M/2$ is required in the worst case [4], which means that as much as half the system's processing capacity could be lost. The other approach is to assign some tasks shares that are less than their desired weights so that no processor is over-utilized. Although this approach may not be able to guarantee each task its weight, the system's overall capacity does not have to be restricted, which is a significant advantage in computationally-intensive systems like Whisper and ASTA. Moreover, allowing task shares to be somewhat malleable circumvents any bin-packing-like intractabilities that might otherwise arise—with frequent weight changes, such intractabilities would have to be dealt with *frequently* at *run-time*. Note that we are still able to offer some service guarantees (albeit weaker than PD^2-OF) with this approach, as discussed later in Sec. 2.2. (In particular, for applications where W in Table 1 is low, the resulting share guarantees may be acceptable.) For these reasons, we use this approach in the schemes we propose. To the best of our knowledge, we are the first to suggest using such an approach to schedule dynamically-changing multiprocessor workloads. The fundamental limitation of partitioned schemes noted above is formalized below.

Theorem 1. *For any partitioned scheduling algorithm, any value ϵ, where $0 < \epsilon < 0.5$, and any integers M and k such that $M \geq 2$ and $k \geq M + 1$, there exists an M-processor task system τ with k tasks such that at least one processor must be initially assigned tasks with total weight at least $1 + W - \epsilon$, where W is the weight of the $\left(M \cdot \lfloor \frac{1}{X} \rfloor + 1\right)^{st}$ heaviest task and X is the weight of the heaviest task.*[2]

Proof. Let the M heaviest tasks have weight $X = 1 - \epsilon$, and let the $(M+1)^{st}$ heaviest task have weight $W = min(M \cdot \epsilon - \delta, 1 - \epsilon)$, where $\delta < \epsilon$. Let the total weight of the remaining $k - (M + 1)$ tasks be δ. (For example, if $\epsilon = 1/3$, $k = 3$, and $M = 2$, then the system consists of three tasks of weight $2/3$.) At least one processor is initially assigned two tasks with total weight at least $1 - \epsilon + W$, and thus is over-utilized by $W - \epsilon$. Since $\epsilon < 0.5$, $M \cdot \lfloor \frac{1}{X} \rfloor + 1 = M + 1$. Hence, W is the weight of the $(M \cdot \lfloor \frac{1}{X} \rfloor + 1)^{st}$ heaviest task, as required. □

2.2 Elements of Repartitioning

In the remainder of this section, we develop the PAS algorithm. PAS is a derivative of the earliest-eligible-virtual-deadline-first (EEVDF) algorithm of Stoica *et al.* [8], with three major differences. First, PAS is designed for multiprocessor systems. Second,

[2] This theorem can be easily extended to the case where $\epsilon \geq 0.5$; however, due to space constraints we omit this extension and its proof.

PAS can enact weight changes with constant drift. (EEVDF can do so only by severely limiting the situations under which tasks may reweight.) Third, PAS can be employed along with any of several approaches for minimizing overall overload error.

Under PAS, a task T requests processing time of an arbitrary size. The size of the i^{th} request of task T is denoted $req(T, i)$. A task is considered *active* if it has an unsatisfied request, and is *passive*, otherwise. The set of active tasks at time t is denoted $\mathcal{A}(t)$. PAS schedules the tasks on each processor on an earliest-deadline-first basis. Moreover, when a task is scheduled under PAS, it is guaranteed at least q units of computation time, where q denotes the scheduling quantum size; however, a task may relinquish its processor within a quantum thus allowing another task to execute. Since we allow a task's weight to vary with time, we denote the weight of a task T at time t as $wt(T, t)$. A task T *reweights* at time t if $wt(T, t - \epsilon) \neq wt(T, t)$ where $\epsilon \to 0^+$. Furthermore, a task T's weight at time t is required to satisfy the following property.

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

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

Any partitioned-based reweighting scheme must address four concerns: **(i)** assigning tasks to processors; **(ii)** determining the processor share of each task; **(iii)** determining the conditions that necessitate a repartitioning; and **(iv)** scheduling tasks in accordance with their assigned shares. We consider each in turn.

Assigning tasks to processors. The problem of assigning tasks to processors is equivalent to the NP-hard bin-packing problem. Given that reweighting events may be frequent, an optimal assignment of tasks to processors is not realistic to maintain. In PAS, we partition N tasks onto M processors in $O(M + N \log N)$ time by first sorting them by weight from heaviest to lightest, and by then placing each on the processor that is the "best fit" (this partitioning method is called *descending best-fit*). We chose this method because it falls within a class of bin-packing heuristics called *reasonable allocation decreasing*, which has been shown by Lopez *et al.* to produce better packings than other types of heuristics [6]. Most importantly, the "descending best-fit" strategy can guarantee that no processor is over-utilized by more than W, where W is the weight of the $\left(M \cdot \lfloor \frac{1}{X} \rfloor + 1\right)^{st}$ "heaviest" task and X is the weight of the "heaviest" task, which is the same limit stated in Thm. 1. Also, under this strategy no processor is over-utilized by more than the weight of the lightest task assigned to it.

Determining task shares. We now consider the problem of determining task shares on over-utilized processors. As mentioned earlier, we have chosen to restrict the shares of such tasks rather than rejecting tasks from the system. However, it is not immediately obvious how to best assess the overall error that results from overload. (Note that the notion of "overload error" is the same as defined earlier. The issue here is how to assess the overall impact of the various overload errors experienced by different tasks.) We consider four different metrics for doing this, and for each, we define a method for determining task shares based on that metric. As a shorthand, we use $sh(T, t)$ to denote task T's share at time t. A summary of the four metrics is given in Table 2. In describing these metrics, we assume that P is an over-utilized processor at time t, T is a task assigned to it at t, and n is the number of such tasks. Many of the claims that are stated below are true only if P is not over-utilized by more than the weight of its lightest assigned task, so we assume this as well. (However, such claims can be easily adjusted to accommodate P being over-utilized by more than the weight of its lightest task.)

The first two metrics are based on absolute differences between weights and shares. The *maximal absolute overall error* (*MAOE*) is given by $max\{wt(T, t) - sh(T, t)\}$. To minimize this metric, the expressed difference should be the same for every task. For example, if five tasks are assigned to a processor that is over utilized by 0.2, then each task's share should be 0.2/5 less than its weight. The other absolute metric, the *average absolute overall error* (*AAOE*), is given by $\sum_T (wt(T, t) - sh(T, t))/n$. It is easy to show that this metric is minimized whenever all task shares sum to one. Since this would be ensured by any reasonable share-assignment strategy, this metric is not interesting.

The next two metrics are based on relative differences between weights and shares. The *maximal relative overall error* (*MROE*) is given by $max\{(wt(T, t) - sh(T, t))/wt(T, t)\}$. This metric is minimized when all task shares are scaled by the same value. For example, if a set of tasks over-utilizes a processor by 0.2, then each task's share would be 1/1.2 times its weight. This scaling is the same as the *proportional-share* scaling used in EEVDF [8]. The final metric is the *average relative overall error* (*AROE*), which is given by $[\sum_T (wt(T, t) - sh(T, t))/wt(T, t)]/n$. This metric is minimized when the heaviest task's share is less than its weight by the amount by which P is over-utilized, and the share of every other task equals its weight. For example, if four tasks A, B, C, and D with weights 0.5, 0.2, 0.2, and 0.2, respectively, are assigned to a processor, then A's share would be $0.5 - 0.1$ (the processor is over-utilized by 0.1), and B, C, and D would each have a share of 0.2.

If task shares are chosen to minimize the MROE metric, then the AROE metric can be shown to be within $(\sqrt{n \cdot wt(T, t)} - 1)^2/(n \cdot wt(T, t))$ of the optimal value stated for it, where T is the heaviest task. Additionally, if task shares are chosen to minimize the AROE metric, then the MROE metric can be shown to be within $(n \cdot wt(T, t) - 1)(n - 1)/(n \cdot wt(T, t))$ of its optimal value, where T is the heaviest task. (Due to page limitations, these formulas are derived only in the full version of this paper.)

In the share-calculation methods described so far, the loss to system utility is measured solely based on the difference (be it absolute or relative) between a task's weight and share. However, in some applications, such a value may not truly capture the loss of utility. For example, suppose that Whisper were implemented so that when hand and feet positions cannot be precisely calculated in time, these positions can be estimated based on the position of the user's head. Then, there could be a great loss of utility if the tasks monitoring the head receive insufficient shares, but much less loss if the tasks monitoring the hands and feet do. In such a case, it may be desirable for the application developer to

Metric Name	Metric Formula	Optimal Share Assignment
MAOE	$max\{T \text{ on } P : \mathit{wt}(T,t) - sh(T,t)\}$	$sh(T,t) = \mathit{wt}(T,t) - \omega(P,t)/n$
AAOE	$\sum_{T \text{ on } P}(\mathit{wt}(T,t) - sh(T,t))/n$	$\sum_{T \text{ on } P} sh(T,t) = 1$
MROE	$max\left\{T \text{ on } P : \frac{\mathit{wt}(T,t) - sh(T,t)}{\mathit{wt}(T,t)}\right\}$	$sh(T,t) = \frac{\mathit{wt}(T,t)}{\sum_{K \in P} \mathit{wt}(K,t)}$
AROE	$\sum_{T \text{ on } P}\left(\frac{\mathit{wt}(T,t) - sh(T,t)}{\mathit{wt}(T,t)}\right)/n$	$sh(T,t) = \begin{cases} \omega(P,t) & \text{if } T \text{ is the heaviest task on } P \text{ at } t \\ \mathit{wt}(T,t) & \text{otherwise} \end{cases}$

Table 2: Four metrics for assessing overload-based error. $\omega(P,t)$ denotes $\left(\sum_{T \text{ on } P} \mathit{wt}(T,t)\right) - 1$, and n is the number of tasks assigned to P at time t. The optimal share assignments apply if P is over-utilized at t, and is not over-utilized by more than the weight of the lightest task on P.

formalize the utility loss as a function of the weight and share of each task. This formalization could potentially be used to determine shares by solving an optimization problem. As we will see shortly, PAS is flexible enough to be able to use such share values (though a few subtle issues do arise in this case).

Repartitioning. As tasks are reweighted, the likelihood of processors becoming *substantially* over-utilized increases dramatically, creating significant overall error (however assessed) on these processors. The extent of overall error can be controlled by repartitioning the system. In order to give the user control over migration overhead, we introduce two methods of repartitioning: **(i)** α-*partitioning* and **(ii)** k-*task-partitioning*; α and k are user-defined values, as discussed below. Both alternatives function in a similar manner: if some tasks reweight and this causes some user-defined condition to be violated, then the system is reset. When the system is reset, the set of tasks is repartitioned (using the descending best-fit method described earlier) and all active tasks issue a new request. Under α-partitioning, the system is reset if any processor is over-utilized by at least α. Under k-task-partitioning, the system is reset if a processor is over-utilized by at least the weight of the k^{th} lightest task on that processor. Note that if some tasks accumulate too much overall error over time, then it may be desirable to trigger a reset, and when the system is repartitioned, use a slightly modified descending best-fit algorithm that discourages the assignment of these tasks to over-utilized processors.

2.3 Scheduling and Reweighting

In this section, we describe how PAS schedules and reweights tasks. To simplify the discussion, we assume that task shares are determined by the MROE metric; later, we explain the adjustments necessary to determine task shares by any metric. Recall that under the MROE metric the share of a task T on a over-utilized processor P at time t is given by

$$sh(T,t) = \frac{\mathit{wt}(T,t)}{\sum_{K \in \mathcal{A}(t,P)} \mathit{wt}(K,t)}, \tag{2}$$

where $\mathcal{A}(t,P)$ is the set of tasks that are active at t on P. PAS schedules tasks in accordance with (2) *even when P is under-utilized*. Thus, PAS will fully utilize any processor to which a task has been assigned. Such a property is advantageous in systems like Whisper and ASTA, which can use more processor time to refine computations. To assess allocation accuracy, we consider the *true ideal allocation of a task T up to time t*, given by

$$\mathit{true\_ideal}(T,t) = \int_0^t sh(T,u)du. \tag{3}$$

As a shorthand we denote the true ideal allocations of the i^{th} request of task T up to time t as $\mathit{true\_ideal}(T,t,i)$, which is formally defined as $\int_{r(T,i)}^{t} sh(T,u)du$, where $r(T,i)$ is the release time of the i^{th} request of task T, formally defined below. We denote the actual allocation of T up to time t by $S(T,t)$, and use $S(T,t,i)$ to represent the amount of T's i^{th} request completed by time t.

Before continuing, we introduce an additional notion of weight that is useful when reweighting tasks. When a task changes weight, there can be a difference between when it initiates the change and when the change is enacted. The time at which a weight change is *initiated* is a user-defined time; the time at which the change is *enacted* is dictated by a set of conditions discussed shortly. If these points in time differ, the old weight is used in between. So that we can distinguish a task's desired weight from that actually used in such an interval, we define the *scheduling weight of a task T at time t*, denoted $\mathit{swt}(T,t)$, as $\mathit{wt}(T,u)$, where u is the last time at or before t that a weight change was enacted for T. We define the *scheduling(-weight-based) ideal allocation of a task T up to time t* as $\mathit{sched\_ideal}(T,t) = \int_0^t \frac{\mathit{swt}(T,u)}{\sum_{K \in \mathcal{A}(u,P)} \mathit{swt}(K,u)} du$. As a shorthand we denote the *scheduling ideal allocation of the i^{th} request of task T up to time t* as $\mathit{sched\_ideal}(T,t,i)$, which is formally defined as $\int_{r(T,i)}^{t} \frac{\mathit{swt}(T,u)}{\sum_{K \in \mathcal{A}(u,P)} \mathit{swt}(K,u)} du$.

Releases and deadlines. Under PAS, it is possible for the deadline of a request to vary with time. Hence, we denote the deadline of the i^{th} request of task T at time t as $d(T,i,t)$, and as a shorthand, we use $d(T,i)$ to denote the time u such that $u = d(T,i,u)$. The *release* $r(T,i)$ and *deadline* $d(T,i,t)$ (at time t) of the i^{th} request of task T are derived as follows, where $ar(T)$ is the arrival time of the first request of T and $\mathit{id\_rem}(T,t,i)$ is the remaining computation of the i^{th} request of task T at t in the scheduling ideal system, defined as $\mathit{id\_rem}(T,t,i) = req(T,i) - \mathit{sched\_ideal}(T,t,i)$.

$$r(T,1) = ar(T) \tag{4}$$

$$d(T,i,t) \;=\; t + \frac{\sum_{K \in \mathcal{A}(t,P)} \mathit{swt}(K,t)}{\mathit{swt}(T,t)} \cdot \mathit{id\_rem}(T,t,i) \quad (5)$$

$$r(T,i+1) \;=\; d(T,i) \quad (6)$$

In the expression added to t to determine $d(T,i,t)$, the first term is a scaling factor, which is the reciprocal of T's share, computed using scheduling weights.

Reweighting. We now introduce two new PAS reweighting rules that are PAS extensions of the PD^2-OF reweighting rules presented by us previously [3]. These rules work by modifying future release times and deadlines and are quite different from reweighting rules considered perviously for EEVDF-based schemes.

Suppose that task T initiates a weight change from weight w to weight v at time t_c. Let i be the request of T satisfying $r(T,i) \leq t_c < d(T,i)$. If $\mathit{req}(T,i) - S(T,t,i) > 0$, then let $\mathit{ac\_rem}(T,t,i) = \mathit{req}(T,i) - S(T,t,i)$; otherwise, $\mathit{ac\_rem}(T,t,i) = \mathit{req}(T,i+1)$. Note that $\mathit{ac\_rem}(T,t,i)$ denotes the actual remaining computation in T's current request or the size of T's next request if the current request has been completed. The *lag of the i^{th} request of task T at time t* is defined as $\mathit{lag}(T,t,i) = \mathit{sched\_ideal}(T,t,i) - S(T,t,i)$. T's lag is positive (negative) if its actual allocation is behind (ahead) its scheduling ideal allocation. The choice of which rule to apply depends on T's lag at time t_c. We say that task T is *positive changeable at time t_c from weight w to v* if $\mathit{lag}(T,t_c,i) \geq 0$, and *negative changeable at time t_c from weight w to v*, otherwise. Because T initiates its weight change at t_c, $\mathit{wt}(T,t_c) = v$ holds; however, T's scheduling weight does not change until the weight 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 we will shortly discuss, any "error" associated with skipping a reweighting event like this is accounted for when determining drift.

Rule P: If T is positive-changeable at time t_c from w to v, then one of the two actions is taken: **(i)** if $\mathit{ac\_rem}(T,t_c,i)/v \leq \mathit{id\_rem}(T,t_c,i)/w$, then T's current request i is halted, its weight change is enacted, and a new request of size $\mathit{ac\_rem}(T,t_c,i)$ is issued for it with a release time of t_c; **(ii)** otherwise, no action is taken until time $d(T,i)$, at which point the weight change is enacted (*i.e.*, the scheduling weight does not change until the end of the current request).

Rule N: If T is negative-changeable at time t_c from w to v, then one of two actions is taken: **(i)** if $v > w$, then T's current request is halted, its weight change is enacted, and a new request of size $\mathit{ac\_rem}(T,t_c,i)$ is issued for it with a release time equal to the time t at which $\mathit{lag}(T,t,i) = 0$ holds; **(ii)** otherwise, the weight change is enacted at time $d(T,i)$.

Intuitively, Rule P changes a task's weight by halting its current request and issuing a new request of size $\mathit{ac\_rem}(T,t_c,i)$ with the new weight, if doing so would improve its scheduling priority. Note that at time t the i^{th} request of task T has a higher scheduling priority than the j^{th} request of task K if $\frac{\mathit{id\_rem}(T,t,i)}{\mathit{swt}(T,t)} \leq \frac{\mathit{id\_rem}(K,t,j)}{\mathit{swt}(K,t)}$. Hence, if $\frac{\mathit{ac\_rem}(T,t_c,i)}{v} \leq \frac{\mathit{id\_rem}(T,t_c,i)}{w}$, then halting T's current request and issuing a new request of size $\mathit{ac\_rem}(T,t_c,i)$ will either improve or maintain T's scheduling priority. A (one-processor) example of a positive-changeable task is given in Fig. 1(a). The depicted example consists of four tasks: T:1/2, which leaves the system at time 2, K:1/6, W:1/6, and V, with an initial weight of 1/6 that increases to 4/6 at time 2. Note that, since K, W, and V have the same deadline, we have arbitrarily chosen V to have the lowest priority. In inset (a), V is positive-changeable since at time 2 it has not yet been scheduled. Note that halting V's current request and issuing a new request of size one improves V's scheduling priority, *i.e.*, $\frac{\mathit{ac\_rem}(V,2,1)}{4/6} = \frac{6}{4} < 4 = \frac{\mathit{id\_rem}(V,2,1)}{1/6}$. Notice that the second request of V is issued 6/4 quanta after time 2. This spacing is in keeping with a new request of weight 4/6 issued at time 2.

Rule N changes the weight of a task by one of two approaches: **(i)** if a task *increases* its weight, then Rule N adjusts the release time of its next request so that it is commensurate with the new weight; **(ii)** if a task *decreases* its weight, then Rule N waits until the end of its current request and then issues the next request with a deadline that is commensurate with the new weight. A (one-processor) example of a negative-changeable task that increases its weight is given in Fig. 1(b). The depicted example consists of the same tasks as in (a), except that we have chosen V to have the highest priority. Notice that the second request of V is issued at time 3, which is the time such that $\mathit{lag}(V,3,1) = \int_0^3 \frac{\mathit{swt}(V,u)}{\sum_{K \in \mathcal{A}(u,p)} \mathit{swt}(K,u)} du - S(V,3,1) = 1 - 1 = 0$. Note that by (5) and (6) the deadline (release time) of the i^{th} ($(i+1)^{st}$) request of a task T is given by $r(T,i) + \mathit{req}(T,i)/(\mathit{swt}(T,r(T,i)))$, assuming all scheduling weights sum to 1.0. Hence, if a task of weight v were to issue a request of size $\mathit{id\_rem}(T,t_c,i)$ at time t_c, then the release time of its next request would be $t_c + \mathit{id\_rem}(T,t_c,i)/v$. (Note that if the scheduling weights do not sum to 1.0, then the deadline must be adjusted accordingly.) A (one-processor) example of a negative-changeable task that decreases its weight is given in Fig. 1(c). The depicted example consists of the same four tasks except that V has an initial weight of 4/6 and decreases its weight at time 1, and T joins the system as soon as V's weight change is enacted.

Theorem 2. *Let $d(T,i)$ be the deadline of the i^{th} request of a task T in a PAS-scheduled system with a quantum size of q where tasks are reweighted by Rules P and N. Then, this request is fulfilled by time $d(T,i) + q$.*[3]

Proof Sketch. Barring reweighting events that force migration, PAS is used independently on each processor. When migrations do occur, the system introduces tasks onto each processor in a manner in keeping with a valid uniprocessor PAS schedule. Hence, we can reduce the correctness of multiprocessor PAS to that of uniprocessor PAS. Since PAS is an EEVDF-derived algorithm,

[3] Deadline tardiness is acceptable, as long as tardiness bounds are reasonably small in comparison to the expected interval length between reweighting events. Otherwise, a task could be repeatedly reweighted before getting a chance to execute with its old weight, which is clearly problematic. Fortunately, in most systems, the quantum size is a settable parameter.

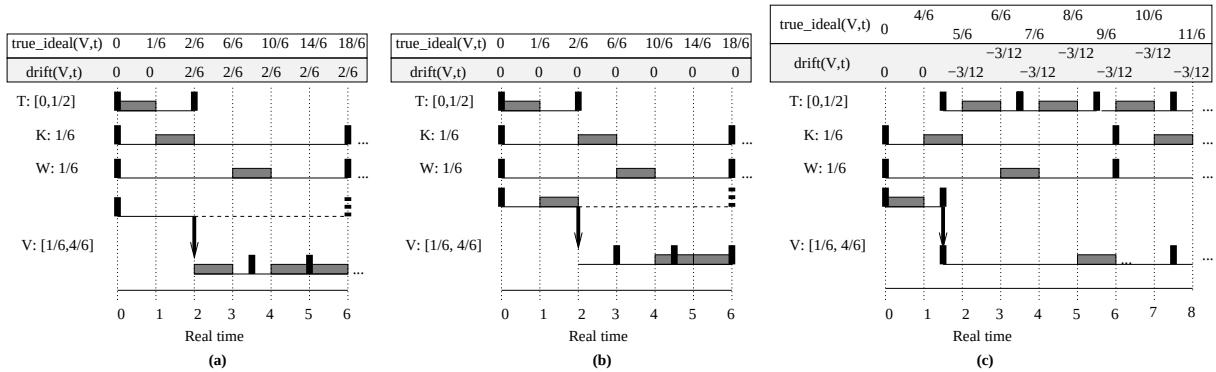

Figure 1: A one-processor system consisting of four tasks, T:$[0, 1/2]$, K:$1/6$, W:$1/6$, and V:$[1/6, 4/6]$. The dotted lines represent the interval up to V's next deadline, which due to reweighting has been changed (as indicated by the solid arrow). The drift and true ideal allocation for V are labeled as a function of time across the top. **(a)** The PAS schedule for the scenario where T is in the system initially and leaves at time 2, V has an initial weight of 1/6 and increases to 4/6 at time 2, and V has a lower priority than both K and W. Since V is not scheduled by time 2, it has positive lag and changes its weight via Rule P, causing the deadline of its current task to become 9/2 and its drift to become 2/6. **(b)** The same scenario as in (a) except that V has higher priority than both K and W. Since V has been scheduled by time 2, it has negative lag and changes its weight via Rule N, causing its next request to have a release time of 3 while maintaining a drift of zero. **(c)** T joins the system at time 6/4 and V has an initial weight of 4/6 that decreases to 1/6 at time 1. Since V has negative lag at time 1, it is changed via Rule N, causing V's next request to have a deadline of 15/2 and V to have a drift of $-3/12$. Note that all requests are of size one.

we can thus use the same proof techniques that Stoica *et al.* [8] used to show that the request of task T under a PAS-scheduled system fulfills its request by $q + d(T, i)$. □

Drift. We now turn our attention to the issue of measuring "drift" under PAS. For most real-time scheduling algorithms, the difference between the true ideal and actual allocation a task receives lies within some bounded range centered at zero. For example, under PAS (without reweighting), the difference between *true_ideal*(T, t) and $S(T, t)$ lies within $(-REQ, REQ)$. When a weight change occurs, the same bounds are maintained, except that they may be centered at a different value. For example, consider again Fig. 1. In inset (a), V's releases and deadlines are commensurate with its new weight starting at time 7/2. Its actual allocation up to this time is 1.0, while its true ideal allocation is $8/6$. Thus, $2/6$ of its true ideal allocation has been permanently "lost." This lost allocation is called its *drift*. Given this loss, barring further reweighting events, the difference between T's true ideal and actual allocations will henceforth be maintained between $-4/6$ and $8/6$ (assuming a maximum request size of one). In general, a task's drift per reweighting event will be nonnegative (nonpositive) if it increases (decreases) its weight. Under PAS, the drift of a task T at time t is formally defined as

$$drift(T, t) = \textit{true\_ideal}(T, u) - S(T, u), \tag{7}$$

where u is the earliest time at which T may issue a new request at or after its most recent weight change.

Theorem 3. *The absolute value of per-event drift under PAS is less than REQ, where system resets* (*i.e., repartitionings*) *are considered reweighting events.*

Proof Sketch. We first show that the absolute value of drift is less than *REQ* drift on a uniprocessor (where obviously no system resets occur). If a task T changes its weight at time t_c via Rule P, then when this weight change is enacted at time t_e (*i.e.*, at t_c under case (i), or at $d(T, i)$ under case (ii)), it is as though an amount of computation equal to *true_ideal*$(T, t_e, i) - S(T, t_e, i)$ is "lost," resulting in drift. (For example, in Fig. 1(a), *true_ideal*$(T, 2, 1) - S(T, 2, 1) = 2/6$, thus that computation is "lost" causing V to drift by 2/6.) Since this value (per reweighting event) is always less than *REQ*, the absolute value of drift is less than *REQ*.

If a task T, during its i^{th} request, changes its weight at time t_c via Rule N and T decreases its weight (case (ii)), then it is as though T leaves with its old weight and rejoins with its new weight at $d(T, i)$. (Stoica, *et al.* proved that a task can leave at a time t if it has equal scheduling ideal and actual allocations.) If T increases its weight (case (i)), then it incurs zero drift since it *immediately* changes the eligibility time of its next request in a manner that is consistent with its new weight. Either way, the absolute value of the drift incurred by this reweighting event is less than REQ. (Notice that in Fig. 1(b), V's drift is 0, while in (c), it is $-3/12$.)

On a multiprocessor, the key is to show that each system reset induces per-task drift in the range $(-REQ, REQ)$. If the first reweighting event is a reset, then each task's drift is bounded by its lag at that time, which lies in the range $(-REQ, REQ)$. The drift due to resets that follow other reweighting events can be calculated similarly, after first accounting for drift introduced by those prior events. □

Unexpected over-utilization. Recall that, according to rules P and N, a reweighting event may be initiated and enacted at different times. It is not difficult to show that this delay is inherent in any reweighting scheme with bounded deadline tardiness. Such delays are particularly problematic when multiple tasks initiate reweighting events simultaneously and weight decreases are necessary to offset weight increases. Consider, for example, Fig. 2, which depicts a one-processor PAS schedule consisting of five

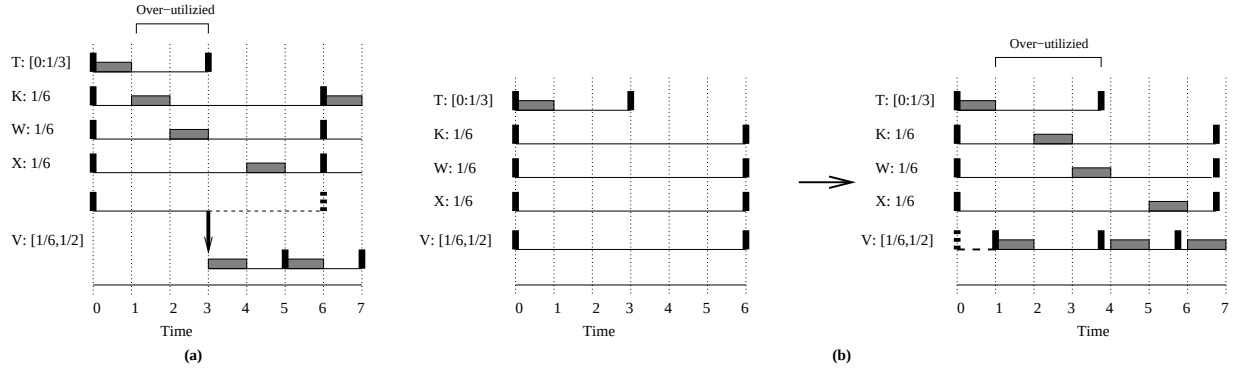

Figure 2: A one-processor system consisting of five tasks, T:$[0, 1/3]$, K:$1/6$, W:$1/6$, X:$1/6$, and V:$[1/6, 1/2]$; T has an initial weight of 1/3 and V has an initial weight of $1/6$. At time 1, task T initiates a weight decrease, and task V would like to increase its weight to 1/2. **(a)** Task V does not initiate its weight increase until T has enacted its weight decrease (at time 3). **(b)** Task V initiates its weight increase at time 1, thus reducing the processor share of every other task in the system. The first part of this inset depicts the schedule before time 1, and the second part depicts the schedule at and after time 1. The period of over-utilization is marked for both insets. Notice that in inset (a) tasks K, W, and X receive more allocations over the range $[0, 7]$ than in inset (b); however task V receives more allocations over the range $[0, 7]$ in (b) than in (a). Also, sum of weights in the system is never more than one, yet the system is over-utilized with respect to all reweighting events not yet enacted of the range $[2, 3]$ in inset (a) and $[2, 11/3]$ in (b).

tasks: T with a weight 1/3 that initiates a weight decrease to zero at time 1; K, W, and X with weight $1/6$; and V with an initial weight of $1/6$ that attempts to increase its weight to $1/2$ at time 1. Note that, by rule P, T cannot enact its weight decrease immediately; hence, if V is allowed to increase its weight, the system will be over-utilized even though all weights sum to one. In these situations, we have two options: **(i)** delay initiating a weight increase until enough weight decreases have been enacted to compensate for the increase (as illustrated in Fig. 2(a)); or **(ii)** let the system become temporarily over-utilized and reduce the share of each task in proportion to its weight (as illustrated in Fig. 2(b)). Note that method (ii) will initiate weight increases sooner than (i); however, method (ii) will take longer than method (i) to enact weight decreases and during this time all tasks that *do not* change their weight will be penalized. Because of these tradeoffs, neither method is inherently superior, and the choice of which method to use in an implementation depends on the target application.

Adjusting PAS for use with any metric. In order to allow PAS to determine task shares via any non-MROE metric, we must make some small changes to the algorithm. Before we continue, note that following property holds for the MROE metric:

QS (queue stability): At any reweighting event on a processor P, the share of each non-reweighting task assigned to P changes by the same multiple: $\frac{old}{new}$, where *old* (*new*) is the total weight of all tasks assigned to P immediately before (after) the reweighting event.

Recall that rules P and N function by changing the future releases and deadlines of a reweighted task. Such a task must be reinserted into the scheduler's priority queue. Since QS guarantees that the shares of all non-reweighted tasks change by the same multiple, these tasks already appear in the queue in the correct order, so they do not have to be reinserted into the priority queue via a rule like P or N. (Moreover, if the concept of *virtual time* is introduced, then the deadlines of such tasks do not have to be recomputed [8].)

For example, suppose that four tasks A, B, C, and D with weights 0.5, 0.2, 0.2, and 0.2, respectively, are assigned to a processor, and at some time, D changes its weight to 0.3. Under the MROE metric, D's weight change causes A's share to change from $0.5/1.1$ to $0.5/1.2$, and the share of both B and C to change from $0.2/1.1$ to $0.2/1.2$. Thus the shares of A, B, and C all change by the same factor, $1.1/1.2$ (the old total weight divided by the new total weight).

Under metrics that are not equivalent to MROE, *QS does not hold*. Consider the same example as above except that the AROE metric is used. Then, A's share changes from $0.5-(1.1-1) = 0.4$ to $0.5-(1.2-1) = 0.3$, while the shares of both B and C remain at 0.2. Thus, A and B (as well as A and C) change shares by a different multiple. As a result, A (or both B and C) must change its (their) share by a rule (*i.e.*, P or N) that reinserts it (them) into the scheduler's priority queue. Due to page limitations, we refer the reader to the full version of this paper for a detailed explanation of how rules P and N can be adapted to explicitly change the share of a task that does not change its weight.

One important consequence of explicitly having to update the share of a non-reweighted task (via rule P or N) is that such a task can incur drift. This problem is not unique to PAS. Indeed, under *any* reweighting scheme, if non-reweighted tasks require share changes by different multiples, then some such tasks will incur drift. (This statement can be justified by extending our prior work [3], which illustrated that drift is fundamental on Pfair-scheduled systems.) Hence, the MROE metric has an inherent advantage since, in the absence of system resets, it can guarantee that the only tasks that incur drift are those that change weight.

Complications with partitioned scheduling. A major complication with the way we have defined partitioned scheduling is handling tasks that *require* share guarantees—we call such tasks *hard* tasks. As discussed earlier, the only way to provide such guarantees is to limit the total weight of all tasks in the system. However, this brings up the question: what if only a hand-

ful of tasks require share guarantees? If the total utilization of all hard tasks is less than $M/4$, where M is the number of processors, then we can guarantee shares for these hard tasks and maintain a utilization cap of M by using the following modified bin-packing algorithm. First, we assign the hard tasks to as few processors as possible without over-utilizing any processor using a "descending first-fit" strategy. Second, we assign the remaining tasks to processors using a "descending best-fit" strategy with one modification—processors that contain hard tasks cannot be over-utilized. With this change, processors that do not contain hard tasks may be over-utilized by more than the weight of their lightest task. However, since the descending first-fit algorithm will assign hard tasks to at most $M/2$ processors, none of the remaining $M/2$ processors will be over-utilized by more than the combined weight of its two lightest tasks. Furthermore, if we know the weight of the heaviest hard task, then we can construct a more permissive bound based on the weight of the heaviest task that can guarantee the share of every hard task and allow the total weight of all hard tasks to exceed $M/4$. (Due to page limitations, we present this bound in the full paper.)

Time complexity. As noted earlier, the time complexity for PAS to partition N tasks onto M processors is $O(M + N \log N)$. If we were to implement PAS using binomial heaps, then the time complexity to make a scheduling decision on a processor P is $O(\log n)$, where n is the number of tasks assigned to P. Recall that when a task changes its weight using either rule P or N, it is reinserted into its processor's priority queue. Thus, $O(\log n)$ time is required to change a task's weight via rule P or N.

As a final comment regarding PAS, we are *not* claiming that it be viewed as the final word regarding partitioned reweighting schemes. However, we have tried hard to devise reasonable approaches for dealing with the fundamental limitation discussed earlier to which such schemes are subject. Thus, we believe that PAS is a good candidate partitioning approach, as claimed earlier.

3 Experimental Results

The results of this paper are part of a longer-term project on adaptive real-time allocation in which both the human-tracking system, Whisper, and the video-enhancement system, ASTA, described in the introduction, will be used as test applications. In this section, we provide extensive simulations of Whisper and ASTA as scheduled by both PD^2-OF and PAS.

Whisper. As noted earlier, Whisper tracks users via speakers that emit white noise attached to each user's hands, feet, and head. Microphones located on the wall or ceiling receive these signals and a tracking computer calculates each speaker's distance from each microphone by measuring the associated signal delay. 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. By varying the number of samples, Whisper can trade measurement accuracy for computation—with more samples, the more accurate and more computationally intensive the calculation. As a signal becomes weaker, the number of samples is increased to maintain the same level of accuracy. As the distance between a speaker and microphone increases, the signal strength decreases. This behavior (along with the use of predictive techniques mentioned in the introduction) can cause task shares changes of up to two orders of magnitude every 10 ms. Since Whisper continuously performs calculations on incoming data, at any point in time, it does not have a significant amount of "useful" data stored in cache. As a result, migration costs in Whisper are fairly small (at least, on a tightly-coupled system, as assumed here, where the main cost of a migration is a loss of cache affinity). In addition, fairness and real-time guarantees are important due to the inherent "tight coupling" among tasks that is required to accurately perform triangulation calculations.

ASTA system. Before describing ASTA in detail, we review some basics of videography. All video is a collection of still images called *frames*. Associated with each frame is an *exposure time*, which denotes the amount of time the camera's shutter was open while taking that frame. Frames with faster exposure times capture moving objects with more detail, while frames with slower exposure times are brighter. If a frame is *underexposed* (*i.e.*, the exposure time is too fast), then the image can be too dark to discern any object. The ASTA system can correct underexposed video while maintaining the detail captured by faster exposure times by combining the information of multiple frames. To intuitively understand how ASTA achieves this behavior, consider the following example. If a camera, **A**, has an exposure time of $1/30^{th}$ of a second, and a second camera, **B**, has an exposure time of $1/15^{th}$ of a second, then for every two frames shot by camera **A** the shutter is open for the same time as one frame shot by **B**. ASTA is capable of exploiting this observation in order to allow camera **A** to shoot frames with the detail of $1/30^{th}$ of a second exposure time but the brightness of $1/15^{th}$ of a second exposure time. As noted earlier, darker objects require more computation than lighter objects to correct. Thus, as dark objects move in the video, the processor shares of tasks assigned to process different areas of the video will change. As a result, tasks will need to adjust their weights as quickly as an object can move across the screen. Since ASTA continuously performs calculations based on previous frames, it performs best when a substantial amount of "useful" data is stored in the cache. As a result, migration costs in ASTA are fairly high. In addition, while strong real-time and fairness guarantees would be desirable in ASTA, they are not as important here as in Whisper, because tasks can function somewhat independently in ASTA.

Experimental system set up. Unfortunately, at this point in time, it is not feasible to produce experiments involving a real implementation of either Whisper or ASTA, for several reasons. First, both the existing Whisper and ASTA systems are single-threaded (and non-adaptive) and consist of several thousands of 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 scheduling and resource-allocation trade-offs involved. The development of PD^2-OF and PAS can be seen as an attempt to articulate these

tradeoffs. Additionally, the focus of this paper is on scheduling methods that facilitate adaptation—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 virtually-reality and multimedia systems that are well beyond the scope of this paper. For these reasons, we have chosen to evaluate the schemes discussed in this paper via simulations of Whisper and ASTA. While just simulations, most of the parameters used here were obtained by implementing and timing the scheduling algorithms discussed in this paper and some of the signal-processing and video-enhancement code in Whisper and ASTA, respectively, on a real multiprocessor testbed. Thus, the behaviors in these simulations should fairly accurately reflect what one would see in a real Whisper or ASTA implementation.

For both Whisper and ASTA, the simulated platform was assumed to be a shared-memory multiprocessor, with four 2.7-GHz processors and a 1-ms quantum. All simulations were run 61 times. Both systems were simulated for 10 secs. We implemented and timed each scheduling scheme considered in our simulations on an actual testbed that is the same as that assumed in our simulations, and found that all scheduling and reweighting computations could be completed within 5 μs. We considered this value to be negligible in comparison to a 1-ms quantum and thus did not consider scheduling overheads in our simulations. We assumed that all preemption and migration costs were the same and corresponded to a loss of cache affinity. We ignored the issue of bus contention, since in prior work, Holman and Anderson have shown that bus contention can be virtually eliminated in Pfair-scheduled systems by *staggering* quantum allocations on different processors [5]. Staggering would be trivial to apply in PAS as well, since in PAS, processors run nearly independently of each other. Based on measurements taken on our testbed system, we estimated Whisper's migration cost as 2 μs–10 μs, and ASTA's as 50 μs–60 μs. While we believe that these costs may be typical for a wide range of systems, in our experiments we vary the migration cost over a slightly larger range.

While the ultimate metric for determining the efficacy of both systems would be user perception, this metric is not currently available, for reasons discussed earlier. Therefore, we compared each of the tested schemes by comparing against the *true* ideal allocation—all references to the "ideal" system in this section refer to this notion of ideal allocation. In particular, we measured the average amount each task is behind its ideal allocation (this value is defined to be nonnegative, *i.e.*, for a task that is not behind its ideal, this value is zero), the maximum amount any task in a task set is behind its ideal, and each task set's "fairness factor." The *fairness factor* of a task set is the largest deviance from the ideal between any two tasks (*e.g.*, if a system has three tasks, one that deviates from its ideal by -10, another by 20, and the third by 50, then the fairness factor is $50 - (-10) = 60$). The fairness factor is a good indication of how fairly a scheme allocates processing capacity. A lower fairness factor means the system is more fair. These metrics should provide us with a reasonable impression of how well the tested schemes will perform when Whisper and ASTA are fully re-implemented. Due to page limitations we are not able to present the results from every possible combination of the methods we presented in Sec. 2. Therefore, we limit our discussion to variants of PAS that use the AROE and MROE metrics, α-partitioning, and handle "unexpected over-utilization" by delaying the initiation of reweighting events.

Profiling the system. PAS can be competitive with PD^2-OF if an appropriate α-value and request size are chosen. To do this, the system must be profiled. We profiled each system by running PAS (for both MROE and AROE) and varying the α-value, request size, and migration cost. Due to space constraints, we will simply state the α-value and request size determined to be the "best" for each simulation, and refer the reader to the full version of the paper for a more in-depth examination of profiling issues.

Whisper experiments. In our Whisper experiments, we simulated three speakers (one per object) revolving around pole in a 1m $\times$ 1m room with a microphone in each corner, as shown in Fig. 3. The pole creates potential occlusions. One task is required for each speaker-microphone pair, for a total of 12 tasks. In each simulation, the speakers were evenly distributed around the pole at an equal distance from the pole, and rotated around the pole at the same speed. The starting position for each speaker was set randomly. 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 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 our testbed system.

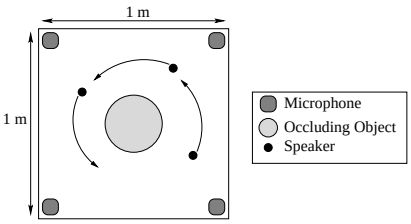

Figure 3: The simulated Whisper system.

In the Whisper 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, the weight of each task changes only once for every 5 cm of distance between its associated speaker and microphone. Sixth, all speakers and microphones are omnidirectional. Finally, all tasks have a minimum weight based on measurements from our testbed system and a maximum weight of 1.0. A task's current weight at any time lies between these two extremes and depends on the corresponding speaker's current position. Even with theses assumptions, frequent share adaptations are required.

We conducted Whisper experiments in which the tracked objects were sampled at a rate of 1,000 Hz, the distance of each object from the room's center was set at 50 cm, the speed of each object was set at 5 m/sec. (such a speed is within the speed of human motion), and the α-value, request size, and migration cost were varied. However, due to page limitations, the graphs below present only a representative sampling our collected data.

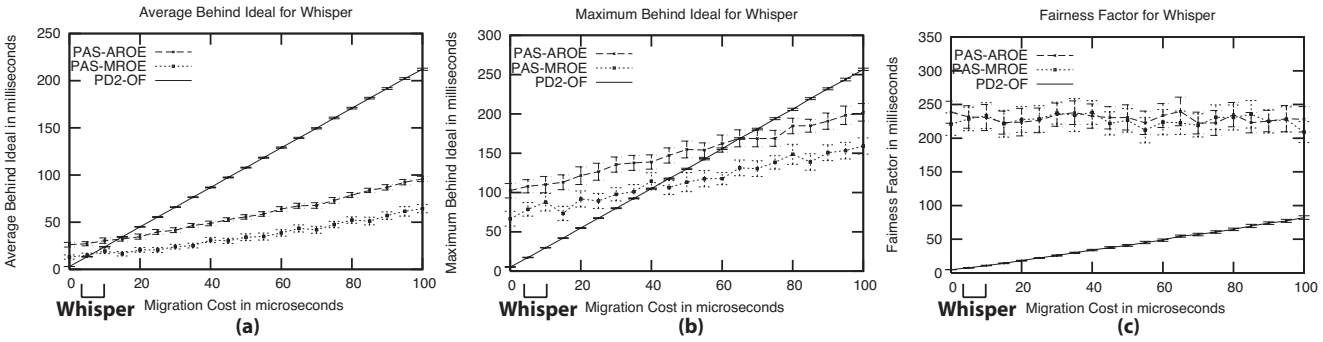

Figure 4: The **(a)** average and **(b)** maximum amount a task is behind its ideal allocation and the **(c)** fairness factor for Whisper as scheduled by PAS using MROE, PAS using AROE, and PD^2-OF. For PAS, the request size is 7 ms and the α-value is 0.1. 98% confidence intervals are shown.

The graphs in Fig. 4 show the results of the Whisper simulations conducted to compare PAS using AROE, PAS using MROE, and PD^2-OF. For both versions of PAS, we used an α-value of 0.075 and a request size of 7 ms. In these experiments, the migration cost was varied from 0 to 100 μs. Insets (a), (b), and (c) depict, respectively, the average and maximum amount by which tasks trail behind their ideal allocations, and the fairness factor, for each scheme, as a function of migration cost. There are four things worth noting here. First, while the performance of each scheme degrades with an increase in migration cost, PD^2-OF degrades much faster. Second, for migrations costs in the range [2 μs, 10 μs], the expected range for Whisper, PAS and PD^2-OF exhibit similar average-case performance, but PD^2-OF is superior in terms of maximum-case error. In addition, the fairness factor of PD^2-OF is *substantially* better. Third, the confidence intervals for the PAS variants in insets (b) and (c) are substantially larger than those for PD^2-OF. This indicates that PD^2-OF's results vary over a much smaller range. Fourth, PAS using MROE performs slightly better than PAS using AROE. This behavior stems from the fact that, under non-MROE metrics, tasks can incur drift even when they do not change their weight and the system is not reset. As a result, under PAS using AROE, more tasks incur drift than in PAS under MROE.

ASTA experiments. In our ASTA experiments, we simulated a 640×640-pixel video feed where a grey square that is 160×160 pixels moves around in a circle with a radius of 160 pixels on a white background. This is illustrated in Fig. 5. The grey square makes one complete rotation every ten seconds. The position of the grey square on the circle is random. Each frame is divided into sixteen 160×160-pixel regions; each of these regions is corrected by a different task. A task's weight is determined by whether the grey square covers its region. By analyzing ASTA's code, we determined that the grey square takes three times more processing time to correct than the white background. Hence, if the grey square completely covers a task's region, then its weight is three times larger than that of a task with an all-white region. The video is shot at a rate of 25 frames per second, and as a result, each frame has an exposure time of 40 ms.

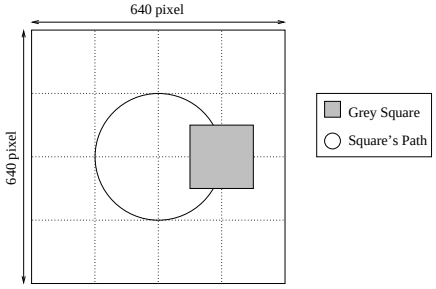

Figure 5: The simulated ASTA system.

The graphs for this set of experiments are shown in Fig. 6. The same information is shown here as for Whisper, with the exception that 0.075 is the α-value for PAS using MROE. There are four things worth noting here. First, as before, while the accuracy of each scheme degrades with an increase in migration cost, PD^2-OF degrades much faster. Second, for migrations costs in the range [50 μs, 60 μs], the expected range for ASTA, both versions of PAS perform *substantially* better than PD^2-OF with respect to the average and maximum metrics. However, PD^2-OF still has a *substantially* better fairness factor. Third, as with Whisper, the confidence intervals for the PAS variants in insets (b) and (c) are substantially larger than for PD^2-OF. This implies that PD^2-OF's results vary over a much smaller range than those of PAS. Fourth, PAS using MROE performs slightly better than PAS using AROE, except in terms of the fairness factor. PAS using AROE has a better fairness factor because PAS using MROE has a slightly lower α-value and as a result the system is reset more often.

These experiments suggest that PAS using MROE is superior to PAS using AROE in terms of both average and maximum error. Furthermore, this behavior will likely be true for any system in which tasks are continually changing weight. However, for systems in which tasks change their weight infrequently, AROE will likely provide better performance in the average case, since the provably superior average case performance of AROE (with static weights) will offset the additional drift that tasks incur. Also note that these experiments suggest that there exist many different scenarios under which PAS and PD^2-OF are each of value. PAS is of value in systems where migration costs are high or where strong real-time and fairness guarantees are not strictly required. However, it has two major drawbacks. First, PAS requires the system to be "profiled" before use. Indeed, if we had chosen an "incorrect" α-value or request size, it is possible that PD^2-OF would have outperformed PAS for any reasonable migration cost. Thus, if the system cannot be profiled beforehand, it is difficult to make any guarantees under PAS. The other drawback of PAS is that, even if both schemes perform well in the average case, the

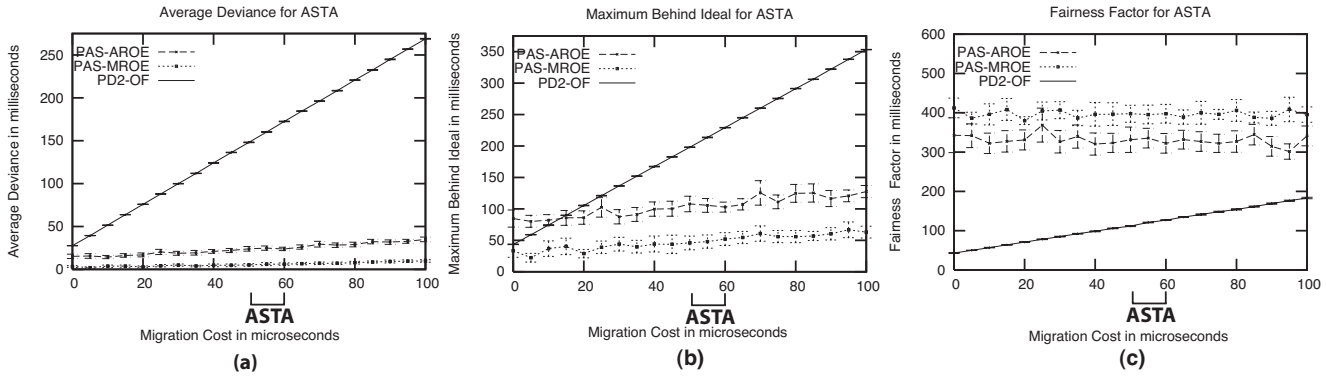

Figure 6: The **(a)** average and **(b)** maximum amount a task is behind its ideal allocation and the **(c)** fairness factor for ASTA as scheduled by PAS using MROE, PAS using AROE, and PD^2-OF. For PAS using AROE, the request size is 7 ms and the α-value is 0.1; for PAS using MROE, the request size is 7 ms and the α-value is 0.075. 98% confidence intervals are shown.

amount by which any one task can deviate from its desired allocation is much harder to predict. On the other hand, in the case of ASTA, PD^2-OF's performance is so poor, it simply is not a viable option, despite its superior real-time and fairness properties. ASTA is a good example of a system for which it is reasonable to trade weaker guarantees for superior performance.

4 Concluding Remarks

We have presented a new multiprocessor reweighting scheme, PAS, which reduces migration costs at the expense of greater allocation error. We have also presented both analytical and experimental comparisons of this scheme with a more accurate but more migration-prone scheme, PD^2-OF. These results suggest that when migration and preemption costs are high, PAS may be the best choice. However, strong real-time and fairness guarantees are not possible under any partitioning-based scheme. Thus, for systems like Whisper, where fairness and timeliness are important and migration costs are low, PD^2-OF is the best choice. However, for systems like ASTA, where migration costs are high and fairness and timeliness are less important, PAS is the best choice.

While our focus in this paper has been on 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 [7]). Both kinds of techniques warrant further study, especially in the domain of multiprocessor platforms.

References

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

[2] E. Bennett and L. McMillan. Video enhancement using per-pixel virtual exposures. *ACM Transactions on Graphics*, 24(3):845–852, 2005.

[3] 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.

[4] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In Joseph Y. Leung, editor, *Handbook on Scheduling Algorithms, Methods, and Models*, pages 30.1–30.19. Chapman Hall/CRC, Boca Raton, Florida, 2004.

[5] P. Holman and J. Anderson. Implementing Pfairness on a symmetric multiprocessor. In *Proceedings of the 10th IEEE Real-time and Embedded Technology and Applications Symposium*, pages 544–553. IEEE, May 2004.

[6] J. Lopez, J. Diaz, and D. Garcia. Utilization bounds for edf scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28(1):39–68, October 2004.

[7] 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. IEEE, December 1999.

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

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