

Accuracy versus Migration Overhead in Real-Time Multiprocessor Reweighting Algorithms*

Aaron Block and James H. Anderson

Department of Computer Science, University of North Carolina at Chapel Hill

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. However, in such schemes preemption and migration overheads can be high. In this paper, we consider the question of whether the lower migration costs of partitioning-based schemes can provide improved average-case performance relative to fair-scheduled systems. Our conclusion is that partitioning-based schemes are capable of providing significantly lower overall error (including “error” due to preemption and migration costs) than fair schemes in the average case. However, partitioning-based schemes are incapable of providing strong fairness and real-time guarantees.

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 real-time workloads on (tightly-coupled) multiprocessor platforms. Such workloads are characterized by the need to change the processor shares¹ 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. (Each is considered in greater detail later in the paper.) 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*, and changes 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 ASTA video-enhancement system [2]. ASTA can improve 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.

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

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

¹In this paper, we consider systems comprised of sequential, recurrent tasks. Each invocation of such a task is called a *job*. A task’s *processor share* (or *utilization*) is the fraction of a single processor’s capacity the task requires in order to meet its timing constraints.

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

Table 1. Summary of worst-case factors. REQ is the maximum request size. W is the weight of the $(M \cdot \lfloor \frac{1}{X} \rfloor + 1)^{st}$ “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, drift can be as high as REQ .

differ.² 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 PD²-OF [3], which is a derivative of the PD² Pfair scheduling 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) scheduling algorithm [8]. PAS is proposed herein as a good candidate partitioned scheduling 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.

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

²In 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 $[0, 1]$.

the weight of the heaviest task. Table 1 shows that algorithms that allow more frequent migrations, like the Pfair-based PD²-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.

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 PD²-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, PD²-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 examine the issue of reweighting in partitioned systems. Because there cannot exist an optimal³ partitioned scheduling algorithm, we focus our attention on different heuristic tradeoffs that can minimize different sources of error. (Due to page limitations, we present only a sampling of the heuristics we have developed. Additional heuristics can be found in the full version of this paper, at <http://www.cs.unc.edu/~anderson/papers.html>.) Before we discuss these tradeoffs in detail, we first consider a fundamental limitation of all partitioning algorithms.

2.1 A Limitation 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 periodic⁴ tasks with an execution cost of 2.0 and a period of 3.0. Two

³A reweighting algorithm is *optimal* if each task can always be granted a share equal to its weight, provided the sum of all weights is at most the number of available processors.

⁴In the *periodic* task model, each task T is characterized by a period $T.p$ and a per-job execution cost $T.e$: every $T.p$ time units, T releases a new job that requires $T.e$ time units to complete.

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. Such caps are due to connections to bin-packing. 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 PD²-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, real 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 $(M \cdot \lfloor \frac{1}{X} \rfloor + 1)^{st}$ heaviest task and X is the weight of the heaviest task.⁵*

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. \square

2.2 Elements of Repartitioning

We now 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, 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 with any of several approaches for minimizing overall overload error.

⁵This 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.

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. $\mathcal{A}(t)$ denotes the set of active tasks at time t . PAS schedules the tasks on each processor on an earliest-deadline-first basis. 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. $wt(T, t)$ denotes the weight of a task T at time t . A task T *reweights* at time t if $wt(T, t - \epsilon) \neq wt(T, t)$, where $\epsilon \rightarrow 0^+$. We use the notation $T:[x, y]$ to denote that T ’s weight ranges over $[x, y]$, and $T:z$ to denote $T:[z, z]$.

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 $(M \cdot \lfloor \frac{1}{X} \rfloor + 1)^{st}$ “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 two 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 two 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 time 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. (Such claims can be easily adjusted to accommodate P

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

Table 2. Two metrics for assessing overall overload-based error. $\omega(P, t)$ denotes $(\sum_{T \text{ on } P} wt(T, t) - 1)$, and n is the number of tasks assigned to P at time t . The optimal share assignments only apply if $0 < \omega(P, t) \leq T$, where T is the lightest task on P at t .

being over-utilized by more than the weight of its lightest task.)

The metrics we consider are based on the relative differences between weights and shares. The *maximal relative overall error (MROE)*, given by $\max\{(wt(T, t) - sh(T, t))/wt(T, t)\}$, 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 *average relative overall error (AROE)*, given by $[\sum_T (wt(T, t) - sh(T, t))/wt(T, t)]/n$, 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 is $0.5 - 0.1$ (the processor is over-utilized by 0.1), and B, C , and D each have a share of 0.2.

In the share-calculation methods described above, the loss to system utility is measured solely based on the relative difference 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 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 α -*partitioning*: if a reweighting event causes any processor to be over-utilized by at least α , the system is reset. A reset causes the set of tasks to be repartitioned (using the descending best-fit method described earlier) and all active tasks to issue a new request. 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 modified descending best-fit algorithm that

discourages assigning these tasks to over-utilized processors.

2.3 Scheduling and Reweighting

In this section, we describe how PAS schedules and reweights tasks on a single processor. To simplify the discussion, we assume that task shares are determined by the MROE metric. (In the full version of the paper we explain the adjustments necessary to determine task shares by any metric, and we also show that under any non-MROE scheme, tasks that do not change their weight can incur drift, and for this reason, MROE schemes will likely be preferable in most circumstances.) Recall that under the MROE metric the share of a task T on an over-utilized processor P at time t is given by

$$sh(T, t) = \frac{wt(T, t)}{\sum_{K \in \mathcal{A}(t, P)} wt(K, t)}, \quad (1)$$

where $\mathcal{A}(t, P)$ is the set of tasks that are active at t on P . PAS schedules tasks in accordance with (1) *even when P is under-utilized*. Thus, PAS fully utilizes 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

$$true_ideal(T, t) = \int_0^t sh(T, u) du. \quad (2)$$

As a shorthand, we denote the true ideal allocation of the i^{th} request of task T up to time t as $true_ideal(T, t, i)$, 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 .

We now 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. We define the *scheduling weight of a task T at time t* , denoted $swt(T, t)$, as $wt(T, u)$, 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

$$\text{sched_ideal}(T, t) = \int_0^t \frac{\text{swt}(T, u)}{\sum_{K \in \mathcal{A}(u, P)} \text{swt}(K, u)} du.$$

As a shorthand, we denote the *scheduling ideal allocation of the i^{th} request of task T up to time t* as $\text{sched_ideal}(T, t, i)$, which is formally defined as $\text{sched_ideal}(T, t) - \text{sched_ideal}(T, r(T, i))$.

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 $id_rem(T, t, i)$ is the remaining computation of the i^{th} request of task T at time t in the scheduling ideal system, defined as $id_rem(T, t, i) = req(T, i) - \text{sched_ideal}(T, t, i)$.

$$r(T, 1) = ar(T) \quad (3)$$

$$d(T, i, t) = t + \frac{\sum_{K \in \mathcal{A}(t, P)} \text{swt}(K, t)}{\text{swt}(T, t)} \cdot id_rem(T, t, i) \quad (4)$$

$$r(T, i + 1) = d(T, i) \quad (5)$$

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. For example, consider Fig. 1(b). Task V in this figure has an initial weight of $1/6$ that changes to $1/2$ at time 3. (This figure is considered in greater detail later.) Observe that $r(V, 1) = 0$, $d(V, 1, 0) = 0 + 1/(1/6) \cdot 1 = 6$, $d(V, 1, 1) = 1 + 1/(1/6) \cdot 5/6 = 6$, $d(V, 1, 2) = 2 + 1/(4/6) \cdot 4/6 = 3$, and $d(V, 1, 3) = 3 + 1/(4/6) \cdot 0 = 3$. Because $d(V, 1, 3) = 3$, we also have $d(V, 1) = 3$ and $r(V, 2) = 3$.

Reweighting. We now introduce two new PAS reweighting rules that are PAS extensions of the PD²-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. (The rules below are applied on a single processor; reweighting events that trigger a repartitioning are dealt with as discussed earlier.)

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 $req(T, i) - S(T, t, i) > 0$, then let $ac_rem(T, t, i) = req(T, i) - S(T, t, i)$; else, $ac_rem(T, t, i) = req(T, i + 1)$. Note that $ac_rem(T, t, i)$ 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 $lag(T, t, i) = \text{sched_ideal}(T, t, i) - S(T, t, i)$. 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 $lag(T, t_c, i) \geq 0$, and *negative changeable*

at time t_c from weight w to v , otherwise. Because T initiates its weight change at t_c , $wt(T, t_c) = v$ 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 discussed below, 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 $ac_rem(T, t_c, i)/v \leq id_rem(T, t_c, i)/w$, then T 's current request i is halted, its weight change is enacted, and a new request is issued with a release time of t_c and a size of $ac_rem(T, t_c, i)$; **(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 $ac_rem(T, t_c, i)$ is issued with a release time equal to the time t at which $lag(T, t, i) = 0$ 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 $ac_rem(T, t_c, i)$ with the new weight, if doing so would improve its scheduling priority. Note that, by (4), at time t the i^{th} request of task T has a higher scheduling priority than the j^{th} request of task K if $\frac{id_rem(T, t, i)}{\text{swt}(T, t)} \leq \frac{id_rem(K, t, j)}{\text{swt}(K, t)}$. Hence, if $\frac{ac_rem(T, t_c, i)}{v} \leq \frac{id_rem(T, t_c, i)}{w}$, then halting T 's current request and issuing a new request of size $ac_rem(T, t_c, i)$ 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: $T:1/2$, $K:1/6$, $W:1/6$, and V . Task T leaves the system at time 2 and task V has an initial weight of $1/6$ that increases to $4/6$ at time 2. Note that, since K , W , and V have the same initial 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.*, $\frac{ac_rem(V, 2, 1)}{4/6} = \frac{6}{4} < 4 = \frac{id_rem(V, 2, 1)}{1/6}$. Note 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

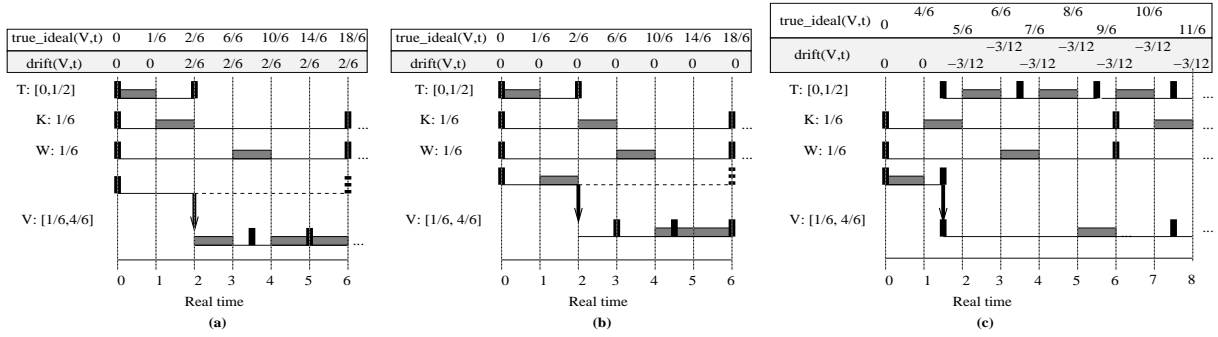


Figure 1. A one-processor system consisting of four tasks, $T:[0, 1/2]$, $K:1/6$, $W:1/6$, and $V:[1/6, 4/6]$. Scheduling is as indicated; for example, in inset (a), T 's first request is released at time 0, has a deadline at time 2, and executed within the interval $[0, 1)$. The dotted lines represent 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 initially 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 initially 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.

consists of the same tasks as in (a), except that we have chosen V to have priority over K and W initially. Note that the second request of V is issued at time 3, which is the time such that $lag(V, 3, 1) = \int_0^3 \frac{swt(V, u)}{\sum_{K \in \mathcal{A}(u, p)} swt(K, u)} du - S(V, 3, 1) = 1 - 1 = 0$. Note also that, by (4) and (5), the deadline (release time) of the i^{th} ($(i + 1)^{st}$) request of a task T is given by $r(T, i) + req(T, i) / (swt(T, r(T, i)))$, assuming all scheduling weights sum to 1.0. Hence, if a task of weight v were to issue a request of size $id_rem(T, t_c, i)$ at time t_c , then the release time of its next request would be $t_c + id_rem(T, t_c, i) / v$. (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 $d(T, i) + q$.⁶

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, we can thus use the same proof techniques that Sto-

ica *et al.* [8] used to show that the i^{th} request of task T in a PAS-scheduled system is fulfilled by $q + d(T, i)$. \square

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

$$drift(T, t) = true_ideal(T, u) - S(T, u), \quad (6)$$

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 on a uniprocessor (where obviously no system

⁶Deadline tardiness is acceptable, as long as tardiness bounds are reasonably small in comparison to the expected interval length between reweighting events. Fortunately, in most systems, the quantum size is a settable parameter.

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 $\text{true_ideal}(T, t_e, i) - S(T, t_e, i)$ is “lost,” resulting in drift. (For example, in Fig. 1(a), $\text{true_ideal}(T, 2, 1) - S(T, 2, 1) = 2/6$, 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 the system with its old weight and rejoins with its new weight at time $d(T, i)$. (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 . (Note 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. \square

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 using the MROE metric. Under non-MROE metrics, $O(n \log n)$ time is required, due to the potential need to re-enqueue non-reweighted tasks.

As a final comment regarding PAS, we do *not* claim that it is 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 PD²-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 share 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. Hence, 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 mainly a loss of cache affinity). Also, 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 $1/30^{\text{th}}$ of a second, and a second camera, **B**, has an exposure time of $1/15^{\text{th}}$ 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 a $1/30^{\text{th}}$ of a second exposure time but the brightness of a $1/15^{\text{th}}$ 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. Hence, 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. Hence, migration costs in ASTA are fairly high. Also, while strong real-time and fairness guarantees are desirable in ASTA, they are not as important as in Whisper, be-

cause 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 PD²-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 μ s. 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 assume that bus contention costs for PAS and PD²-OF are the same, since in prior work, Holman and Anderson have shown that any additional bus contention incurred under PD²-OF can be virtually eliminated by *staggering* quantum allocations on different processors [5]. Based on measurements taken on our testbed system, we estimated Whisper’s migration cost as 2 μ s–10 μ s, and ASTA’s as 50 μ s–60 μ s. While we believe that these costs may be typical for a wide range of systems, in our experiments we varied 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* (FF) 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 FF is $50 - (-10) = 60$). The FF is a good indication of how fairly a scheme allocates processing capacity. A lower FF means the system is more fair. For applications like Whisper, where the output generated by multiple tasks is periodically combined, a low FF is important, since if any one task is “behind,” then the performance of the entire system is impacted; however, for applications like ASTA, where tasks are more independent, a high FF does not affect the system’s performance nearly as much. 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.

Profiling the system. PAS can be competitive with PD²-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. For brevity, we will simply state the α -value and request size determined to be the “best” for each simulation.

Whisper experiments. In our Whisper experiments, we simulated three speakers (one per object) revolving around a pole in a 1m \times 1m room with a microphone in each corner, as shown in Fig. 2. The pole creates potential occlusions. For each speaker/microphone pair, one task is required 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.

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

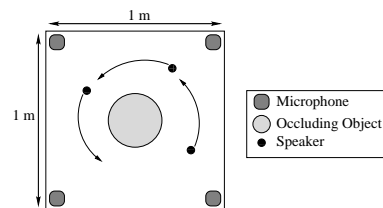


Figure 2. The Whisper system.

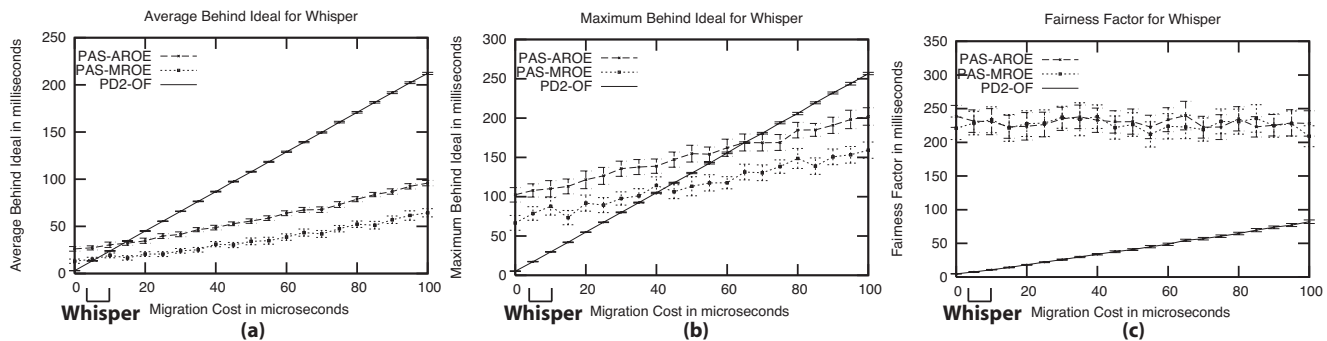


Figure 3. The (a) average and (b) maximum amount a task is behind its ideal allocation and the (c) FF for Whisper as scheduled by PAS (using MROE and AROE), and PD²-OF. For PAS, the request size is 7 ms and the α -value is 0.1. 98% confidence intervals are shown.

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 these 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 migration cost were varied. The graphs below give a representative sampling our collected data.

The graphs in Fig. 3 show the results of the Whisper simulations conducted to compare PAS using AROE, PAS using MROE, and PD²-OF. For both versions of PAS, we used an α -value of 0.1 and a request size of 7 ms. In these experiments, the migration cost was varied from 0 to 100 μ s. Insets (a), (b), and (c) depict, respectively, the average and maximum amount by which tasks trail behind their ideal allocations, and the FF, 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, PD²-OF degrades much faster. Second, for migrations costs in the range [2 μ s, 10 μ s], the expected range for Whisper, PAS and PD²-OF exhibit similar average-case performance, but PD²-OF is superior in terms of maximum error. In addition, the FF of PD²-OF is *substantially* better. Third, the confidence intervals for the PAS variants in insets (b) and (c) are substantially larger than those for PD²-OF. This indicates that PD²-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. Hence, under PAS using AROE, more tasks incur drift than in PAS under MROE.

ASTA experiments. In our ASTA experiments, we simulated a 640 \times 640-pixel video feed where a grey square that is 160 \times 160 pixels moves around in a circle with a radius of 160 pixels

on a white background. This is illustrated in Fig. 4. 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 \times 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. Hence, each frame has an exposure time of 40 ms.

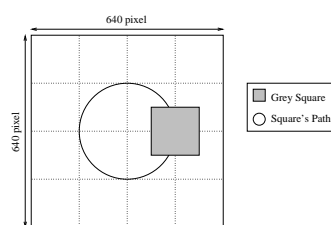


Figure 4. The ASTA system.

The graphs for this set of experiments are shown in Fig. 5. The same information is shown here as for Whisper, with the exception the α -value for PAS using MROE and AROE is 0.075. There are three things worth noting here. First, as before, while the accuracy of each scheme degrades with an increase in migration cost, PD²-OF degrades much faster. Second, for migrations costs in the range [50 μ s, 60 μ s], the expected range for ASTA, both versions of PAS perform *substantially* better than PD²-OF with respect to the average and maximum metrics. However, PD²-OF still has a *substantially* better FF. Third, as with Whisper, the confidence intervals for the PAS variants in insets (b) and (c) are substantially larger than for PD²-OF. This implies that PD²-OF’s results vary over a much smaller range than those of PAS.

Note that these two experimental studies suggest that PAS using MROE is superior to PAS using AROE in terms of both average and maximum error. The reason for this behavior is that, as we mentioned before, under non-MROE algorithms, tasks that do not change their weight can incur drift.

Also note that these experiments suggest that there exist many different scenarios under which PAS and PD²-OF are each of value. PAS is of value in systems where migration costs are

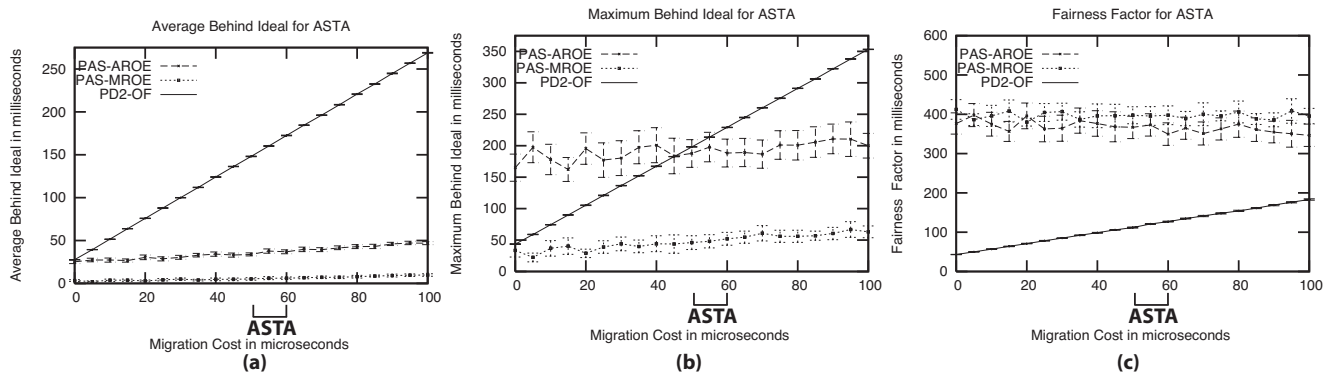


Figure 5. The (a) average and (b) maximum amount a task is behind its ideal allocation and the (c) FF for ASTA as scheduled by PAS (using MROE and AROE), and PD²-OF. For PAS the request size is 7 ms and the α -value is 0.075. 98% confidence intervals are shown.

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 PD²-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 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, PD²-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, PD²-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, PD²-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. Thus, *both algorithms are of value* and will be the best choice in certain application scenarios.

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.