Task Reweighting under Global Scheduling on Multiprocessors *

Aaron Block, James H. Anderson, and UmaMaheswari C. Devi

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

March 2007

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 and that partitioning-based scheduling algorithms can reweight tasks with better average-case performance, but greater error. However, preemption and migration overheads can be high in fair schemes. In this paper, we consider the question of whether non-fair, earliest-deadline-first (EDF) global scheduling techniques can improve the accuracy of reweighting relative to partitioning-based schemes and provide improved average-case performance relative to fair-scheduled systems. Our conclusion is that, for soft real-time systems, global EDF schemes provide a good mix of accuracy and average-case performance.

* Work supported by NSF grants CCR 0204312, CNS 0309825, CNS 0408996, and CCF 0541056. 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, 11, 13, 5]. In addition, *multiprocessor* platforms are of growing importance, due to both hardware trends such as the emergence of multicore technologies and the prevalence of computationally-intensive applications for which single-processor designs are not sufficient. In prior work [3, 5], Block and colleagues considered the use of both fair and partitioning-based algorithms to schedule highly-adaptive workloads on (tightly-coupled) multiprocessor platforms, where the processor shares of tasks change frequently and to a significant extent. Fair scheduling techniques achieve high accuracy in enacting share changes, but do so at the expense of potentially frequent task preemptions and migrations among processors. Partitioning algorithms, in contrast, entail less overhead, but provide poorer (but sometimes acceptable) accuracy. The focus of this paper is adaptive global scheduling algorithms that avoid the high preemption and migration costs of fair scheduling techniques, yet have superior accuracy relative to partitioning-based schemes. The primary drawback of such global scheduling algorithms is that, in order to fully utilize a multiprocessor system, bounded deadline misses must be acceptable. (This utilization-versus-deadline-tardiness issue is an even greater drawback for partitioning schemes: systems exist with total utilization slightly greater than $m/2$ that such schemes cannot correctly schedule on m processors even if bounded tardiness is permissible.) The key issue we address is whether the lower migration/preemption overheads and improved accuracy of such algorithms are sufficient to compensate for their inability to meet all deadlines.

Whisper. To motivate the need for this work, we consider two example applications under development at The University of North Carolina at Chapel Hill (each of which is described in greater detail later in the paper). The first of these is the Whisper tracking system, which performs full-body tracking in virtual environments [14]. 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. 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. The second motivating application is the 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 ten-processor multicore platform will be used.

Dynamic sporadic tasks. In this paper, we are primarily concerned with *dynamic sporadic tasks*. Each such task T_i releases a sequence of *jobs*, $T_i^1, T_i^2, \ldots$. Each task is defined by the *execution cost* of each of its jobs, denoted $\mathsf{e}(T_i^j)$, and its *weight* at any time t, denoted $\mathsf{wt}(T_i, t)$, which specifies the fraction of a single processor it requires. For example, one method for assigning tasks in ASTA is to divide each frame into sixteen regions of equal size, where each region is corrected by a different task as depicted in Fig. 1. Each job of each task processes a portion of one frame (assuming the job's are not "halted" as we will discuss later).

$\mathbf{T_1}$	$\mathbf{T_2}$	$\mathbf{T_3}$	$\mathbf{T_4}$
$\mathbf{T_5}$	$\mathbf{T_6}$	$\mathbf{T_7}$	$\mathbf{T_8}$
$\mathbf{T_9}$	$\mathbf{T_{10}}$	$\mathbf{T_{11}}$	$\mathbf{T_{12}}$
$\mathbf{T_{13}}$	$\mathbf{T_{14}}$	$\mathbf{T_{15}}$	$\mathbf{T_{16}}$

Figure 1: An example frame layout in ASTA, where the responsibilities for correcting a frame is divided among sixteen tasks, each of which corrects a fraction of the screen. T_i denotes the i^{th} task of the system T.

This definition of a task differs from the usual definition of a *sporadic* task, wherein per-job execution costs and weights do not change and relative deadlines are specified for jobs. In our model, execution costs and weights are specified as first-class concepts, and from these, relative deadlines can be derived. 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. We refer to the process of enacting task weight/share changes as *reweighting*.

Summary of results. In this paper, we consider five reweighting-capable scheduling algorithms: a previous fair algorithm developed by Block *et al.* called PD^2-OI [3],[1] which is a derivative of the PD^2 Pfair algorithm [1]; a previous partitioning-based algorithm developed by Block *et al.* called the *partitioned-adaptive scheduling* (PAS) algorithm [5]; the *non-preemptive-partitioned-adaptive-scheduling* (NP-PAS) algorithm, which is a non-preemptive variant of PAS; and two new algorithms proposed herein, the *changeable-earliest-deadline-first* (CNG-EDF) algorithm, which is a derivative of the well known global-earliest-deadline-first (EDF) algorithm, and the *non-preemptive-changeable-earliest-deadline-first* (NP-CNG-EDF) algorithm, which is a non-preemptive variant of CNG-EDF.

[1]In a preliminary version of [3], which appeared in [4], PD^2-OI was referred to as PD^2-OF.

Scheme	Tardiness	Drift	Overload	Migrations	Preemptions
PD^2-OI	0[3]	2	0	every quantum	every quantum
PAS	1	$\mathsf{e}_{\mathsf{max}}(T_i)$	W	weight-change events	weight-change events & job releases
NP-PAS	$\mathsf{e}(T_i^j)$ +$\mathsf{e}_{\mathsf{max}}(T_i)$+1	$\mathsf{e}_{\mathsf{max}}(T_i)$	W	weight-change events	weight-change events
CNG-EDF	$\kappa(m-1)$ +$\mathsf{e}_{\mathsf{max}}(T_i)$	$\mathsf{e}_{\mathsf{max}}(T_i)$	0	weight-change events, job releases & job completions	weight-change events & job releases
NP-CNG-EDF	$\kappa(m)$ +$\mathsf{e}_{\mathsf{max}}(T_i)$	$\mathsf{e}_{\mathsf{max}}(T_i)$	0	only between jobs	never

Table 1: Summary of worst-case results.

Our results are summarized in Table 1, which lists the accuracy, migration cost, and preemption cost of each of the above schemes. Accuracy is assessed in terms of three quantities, "drift," "overload error," and "tardiness," 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*, which arises under partitioning-based schemes (see [5]), is the error that results from a scheduler's inability to allocate a task a share equal to its desired weight. 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 overload error is the difference between the weight and share of these tasks, *i.e.*, $2/3 - 1/2 = 1/6$. *Tardiness* is the maximal amount by which any job can miss its deadline. Of these three types of error, overload error is potentially the most detrimental, since drift is a one-time error assessed per reweighting event and tardiness is bounded[2] in the schemes we consider. Overload error, on the other hand, accumulates over time.

In Table 1, $\mathsf{e}_{\mathsf{max}}(T_i)$ denotes the *maximum execution cost* of any job of task T_i, $\mathsf{wt}_{\mathsf{max}}(T_i)$ denotes the *maximal weight* of task T_i at any time, and W denotes the maximal weight of the $(m \cdot \lfloor 1/X \rfloor + 1)^{st}$ "heaviest" task (by maximal weight) in T, where m is the number of processors, T is the set of all tasks, and X is the maximal weight of the heaviest task. Furthermore,

$$\kappa(\ell) = \frac{\sum_{T_z \in \mathcal{E}_{\max}(T,\, \ell)} \mathsf{e}_{\mathsf{max}}(T_z)}{m - \sum_{T_z \in \mathcal{X}_{\max}(T,\, \ell-1)} \mathsf{wt}_{\max}(T_z)} + \mathsf{e}_{\mathsf{max}}(T_i).$$

where $\mathcal{E}_{\max}(T, \ell)$ is the set of ℓ tasks in T with the highest *maximal* execution cost and $\mathcal{X}_{\max}(T, \ell-1)$ is the set of $\ell - 1$ tasks in T of largest *maximal* weight. (This term is derived from prior work by Devi and Anderson on deadline tardiness under global EDF scheduling [7].) Table 1 shows that algorithms that allow more frequent migrations and preemptions, like PD^2-OI, produce

[2]When a job misses its deadline, the release of its task's next job is not delayed; thus, bounded deadline tardiness is sufficient to ensure accurate share allocations over the long term.

[3]A tardiness of zero is guaranteed to hold for a given task T_i with an execution cost of a and a weight of b/c only if there exists some integer n such that $b \cdot n = a$; otherwise, T_i's tardiness may be up to one quantum.

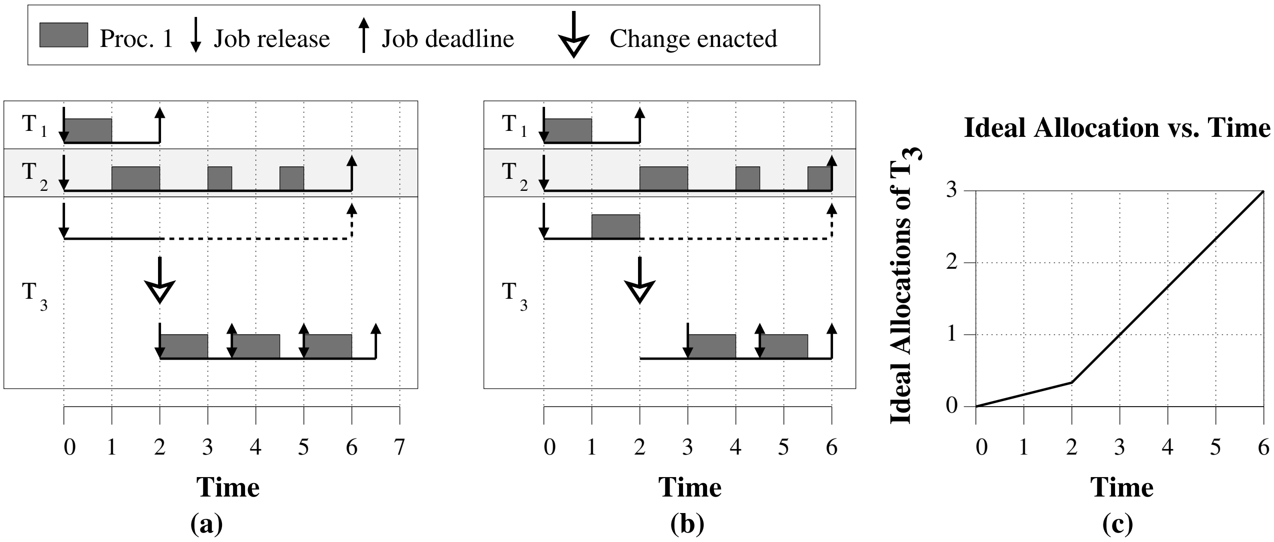


Figure 2: A one-processor system T with three tasks scheduled by EDF (with two reweighting rules) : T_1 with an execution cost of 1 and a weight of $1/2$ that leaves the system at time 2, T_2 with an execution cost of 2 and a weight of $1/3$, and T_3 with an execution cost of 3 and an initial weight of $1/6$ that increases to $4/6$ at time 2. Since the first jobs of T_2 and T_3 have the same deadline, we can arbitrarily choose which job has a higher scheduling priority. Inset **(a)** depicts the case where T_2 has higher priority, and as a result, T_3 is "under-allocated" relative to the ideal, when it reweights at time 2. Thus, when T_3 reweights, its current job "halts" and the next job is immediately released. Inset **(b)** depicts the case where T_3 has higher priority, and as a result, T_3 is "over-allocated" relative to the ideal, when it reweights at time 2. Thus, when T_3 reweights, its current job "halts" and its next job is not released until the difference between T_3's ideal and actual allocations is zero (at time 3). The dotted lines denote the interval of the first job of T_3 that has been changed by the reweighting event. Inset **(c)** depicts the ideal allocation of T_3 as a function of time. Notice that, at time 2, when T_3 increases its weight from $1/6$ to $4/6$, the ideal allocation rate increases from $1/6$ to $4/6$, and that at time 3, T_3's total ideal allocation equals 1.

little drift, no overload error, and no tardiness; however, algorithms that restrict the frequency of migrations and preemptions can produce greater drift, overload error, and/or tardiness.

In all five reweighting algorithms, tasks change weight via one of two rules that are based on whether a task's "active" job is over- or under-allocated relative to an ideal schedule. If a task with an active job reweights when that job is under-allocated, then the change is "enacted" by immediately "halting" the active job and releasing a new job with the remaining execution time. If a task reweights while its active job is over-allocated, then the change is enacted by immediately halting the active job and releasing the next job when the task's "ideal" allocation equals its "actual" allocation. For example, consider the one-processor system depicted in Fig. 2 as scheduled by EDF. The depicted system consists of three tasks: T_1 with an execution cost of 1 and a weight of $1/2$ that leaves the system at time 2; T_2 with an execution cost of 2 and a weight of $1/3$; and T_3 with an execution cost of 1 and an initial weight of $1/6$ that increases to $4/6$ at time 2. Since the first jobs of T_2 and T_3 have the same deadline, they have the same priority under EDF, and we can break this tie in favor of either task. Inset (a) depicts the schedule where the tie is broken in favor of T_2 and inset (b) depicts the schedule where the tie is broken in favor T_3. In inset (a), since T_3 is not scheduled before it is reweighted (at time 2), it is under-allocated relative to the ideal (depicted in inset (c)). Therefore, the

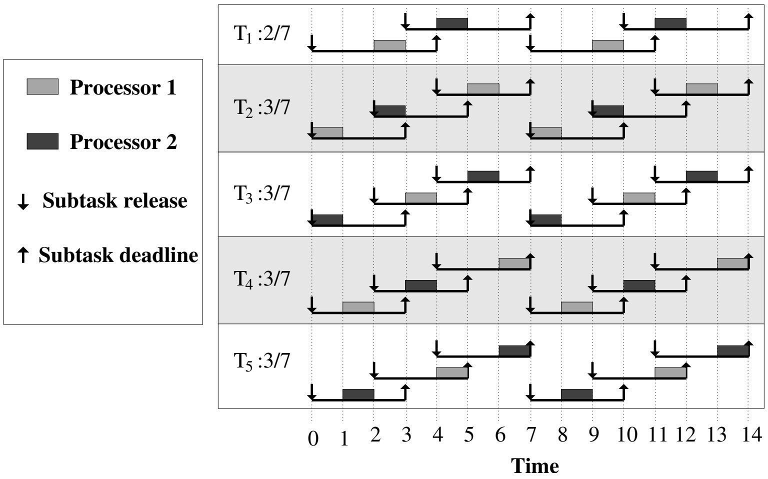


Figure 3: A two-processor system T scheduled by PD^2-OI with five tasks: T_1 with an execution cost of 2 and a weight of $2/7$; T_2 and T_3, each with an execution cost of 1 and a weight of $3/7$; and T_4 and T_5, each with an execution cost of 3 and a weight of $3/7$. Each quantum-length subtask has an *Pfair window*, denoted by the solid line, in which it must be scheduled. In accordance with the PD^2 algorithm, tasks are scheduled in earliest-pseudo-deadline-first order with two tie-breaks (discussed in [12]).

task immediately halts its current job and releases a new job with the new weight. In inset (b), T_3 is over-allocated relative to the ideal at time 2, since by time 2 it has been scheduled for one quantum and has received an ideal allocation of $2/6$. Hence, its weight is immediately increased to $4/6$ and its next job is released when its ideal allocation equals its actual allocation, which is at time 3. It is important to note that in this brief introduction, we have glossed over a number of complexities involved in these two reweighting rules, especially in the definition of the ideal system.

Throughout this paper we often discuss the idea a job "halting." When a job is "halted" this does not mean that it is abandoned, but rather that it is "split" into two jobs. For example, under ASTA setup described by Fig. 1, in the absence of job halting, each job renders one portion of a frame. However, suppose that a job T_i^j that had an execution cost of five were "halted" after completing three units of execution by a weight increase. This does not imply that the portion of the frame being corrected by T_i^j is abandoned, but rather that "work" of T_i^j is split into two jobs: T_i^j that completes the first three units of execution and T_i^{j+1} that completes the remaining two units of execution.

PD^2-OI. We now briefly review the PD^2-OI reweighting algorithm. As mentioned earlier, PD^2-OI is based on the PD^2 Pfair algorithm. PD^2 schedules a task by breaking it into a sequence of *subtasks*, each of which represents one quantum of execution. Associated with each subtask is a *pseudo-release* and *pseudo-deadline* (often called "release" and "deadline" for brevity). PD^2 schedules each subtask on an earliest-pseudo-deadline-first basis with two tie-breaking rules, which are used in the event of a deadline tie. (A more thorough discussion of PD^2 can be found in [12].) PD^2-OI is the same as PD^2, except that PD^2-OI allows the weight of a task to change via the two reweighting rules described earlier. For example, consider Fig. 3, which depicts a schedule without weight changes. (We ignore weight changes in the rest of this description since they are handled in a manner similar to Fig. 2.) The depicted system is scheduled on two processors and consists of five tasks: T_1 with a weight of $2/7$ and an execution cost of 2; T_2 and T_3, each with a weight of $3/7$ and an execution cost of 1; and T_4 and T_5, each with a weight of $3/7$ and an execution cost of 2. Notice that, in this schedule, tasks are scheduled one quantum at a time. As a result, tasks may be preempted at the end of every quantum and may migrate nearly as frequently. Also note that, because tasks are scheduled one quantum at a time, a task's execution cost is assumed to be a multiple of the quantum size (and must be rounded up if this is not the case) and the scheduling of a task depends only on its weight. Notice that in this schedule every subtask is scheduled by its deadline and every task of the same weight receives allocations at the same rate, *e.g.*, T_2 and T_3 receive approximately one allocation in any $7/3$-quantum interval. Finally, another disadvantage of PD^2 (aside from frequent migrations and preemptions) is that a task can have a tardiness of up to one quantum if the execution cost of the task is not a multiple of the numerator of the task's weight (see Footnote 3).

PAS and NP-PAS. We now turn our attention to PAS and NP-PAS. As mentioned earlier, both are partitioning algorithms, and as such, under each algorithm, tasks are assigned to processors for extended durations of time. (The only reason a task may migrate between processors is because of a reweighting event.) On each processor, each task is scheduled on an EDF basis. As discussed earlier, there exist task sets for which any partitioned algorithm must over-utilize a processor. There are two options for handling such a scenario: reject one or more tasks or reduce the shares allocated to some tasks. In PAS and NP-PAS, the latter option is employed. (A thorough discussion of this choice is given in [5].) To achieve this behavior, the deadline of tasks that are assigned to "over-allocated" processors are increased in proportion to the amount by which that processor is over-allocated. For example, if a task T_i normally has a relative deadline of 40 time units and is assigned to a processor that is over-utilized by 25%, then it is scheduled with a relative deadline of $40 + 25\% \cdot 40 = 50$ time units. The difference between PAS and NP-PAS is that under PAS a job can be preempted and under NP-PAS a job cannot be preempted. Under PAS and NP-PAS, a task changes its weight via the two reweighting rules described earlier.

It is important to note that when a reweighting event occurs it is possible that the particular task partitioning before the reweighting event may be a "poor" partitioning after the reweighting event. For example, consider the scenario depicted in Fig. 4, in which four tasks, $T_1, \ldots, T_4$, of weight $1/2$ are assigned to two processors, with T_1 and T_2 on Processor 1 and T_3 and T_4 on Processor 2. At some later time, a reweighting event occurs that causes T_1 and T_2 to reweight to $3/4$ and T_3 and T_4 to reweight to $1/4$. This reweighting event causes Processor 1 to be over-utilized by $1/2$ and Processor 2 to be under-utilized by

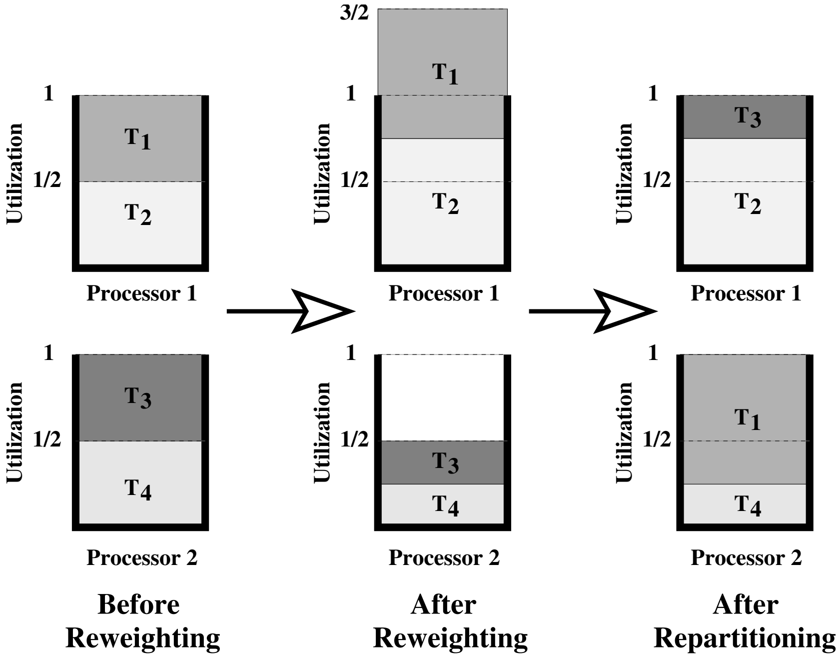


Figure 4: A partitioned two-processor system T with four tasks: T_1 and T_2, each with an initial weight of $1/2$ that at some time t_c changes to $3/4$; and T_3 and T_4, each with an initial weight of $1/2$ that changes $1/4$ at t_c. Initially, T_1 and T_2 are assigned to Processor 1 and T_3 and T_4 are assigned to Processor 2. After the reweighting event occurs at t_c, Processor 1 is over-utilized by $1/2$, and Processor 2 is under-utilized by $1/2$. Thus, the tasks are repartitioned so that neither Processor 1 nor 2 is over-utilized.

$1/2$. In order to prevent either processor from being over-utilized, we can "repartition" the tasks so that neither Processor 1 nor Processor 2 is over-utilized. It is because of this behavior that tasks may migrate under PAS or NP-PAS.

As an example of scheduling under PAS and NP-PAS, consider the system in Fig. 5, which depicts the same two-processor task system as in Fig. 3, except that in Fig. 5(a) the system is scheduled by PAS and in Fig. 5(b) it is scheduled by NP-PAS. In both insets (a) and (b), tasks T_1, T_3, and T_5 are assigned to Processor 1 and tasks T_2 and T_4 are assigned to Processor 2. Notice that the total utilization on Processor 1 is $8/7$. Thus, Processor 1 is over-utilized by $1/7$, and as a result, all tasks assigned to Processor 1 have their deadlines scaled by $8/7$ in comparison to identical tasks assigned to Processor 2. For example, despite the fact that T_4 and T_5 both have the same weight and execution cost, the first deadline of T_4 is at time 7, whereas that of T_5 is at time 8. As a result, PAS and NP-PAS, unlike PD^2-OI, do not have the characteristic that two tasks with the same execution cost and weight receive allocations at approximately the same rate. Finally, notice that, since PAS and NP-PAS allow a job to be scheduled for more than one quantum at a time, jobs in PAS are preempted much less frequently than in PD^2-OI, and jobs in NP-PAS are never preempted (unless the system needs to be repartitioned.)

Comparing these examples, we can see that while PD^2-OI allocates capacity more fairly, it also migrates and preempts tasks much more frequently. As a result, on systems where migration and preemption costs are high, PAS and NP-PAS may be a

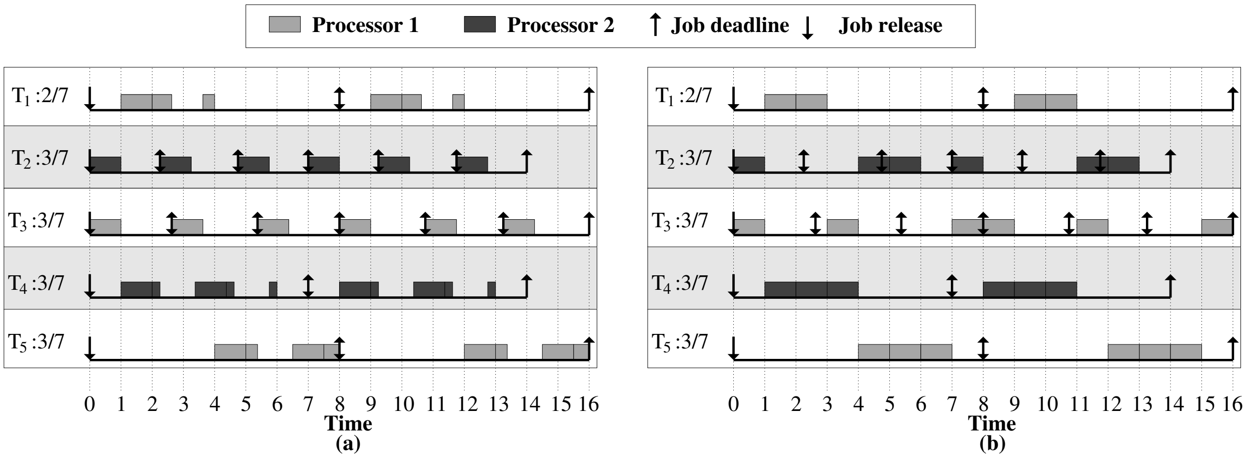


Figure 5: A two-processor system T scheduled by **(a)** PAS and **(b)** NP-PAS with five tasks: T_1 with an execution cost of 2 and a weight of $2/7$; T_2 and T_3, each with an execution cost of 1 and a weight of $3/7$; and T_4 and T_5, each with an execution cost of 3 and a weight of $3/7$. Tasks T_1, T_3, and T_5 are assigned to Processor 1, and tasks T_2 and T_4 are assigned to Processor 2. Each job has an *active window*, denoted by the solid line, in which it must be scheduled. Tasks on each processor are scheduled via EDF. Notice that since Processor 1 is "overloaded," jobs of tasks assigned to Processor 1 have larger deadlines than those assigned to Processor 2.

Scheme	Summary
PD^2-OI	A Pfair-based algorithm that may preempt a task every quantum.
PAS	An EDF algorithm where tasks are partitioned onto processors.
NP-PAS	A non-preemptive variant of PAS.
CNG-EDF	An EDF algorithm where tasks are scheduled from a single global priority queue.
NP-CNG-EDF	A non-preemptive variant of CNG-EDF.

Table 2: Brief description of all major scheduling algorithms considered in this paper.

better choice. For quick reference, a brief summary of each algorithm is provided in Table 2.

Contributions. A key contribution of this paper is to define CNG-EDF and NP-CNG-EDF, which is done by giving reweighting rules for each. These rules indicate when and how reweighting requests are enacted. In addition to devising these reweighting rules, we also establish the error bounds for CNG-EDF and NP-CNG-EDF in Table 1. The question that then remains is: for the five aforementioned algorithms, how do drift, overload error, and tardiness compare to any error due to migration and preemption costs? We attempt to answer this question via extensive simulation studies of Whisper and ASTA. In these studies, real migration and preemption costs were assumed based on actual measured values. These studies confirm the expectation that, while CNG-EDF and NP-CNG-EDF provide a good compromise of accuracy and average-case performance, *there exists no single "best" algorithm*: for each algorithm, application scenarios exist for which that algorithm is the best choice.

The rest of this paper is organized as follows. In Secs. 2 and 3, we discuss the CNG-EDF and NP-CNG-EDF algorithms in greater detail. Then, in Sec. 4, we establish the properties mentioned above. Our experimental evaluation is presented in Sec. 5.

We conclude in Sec. 6.

2 System Model and Scheduling

In this section, we define our system model and the $\mathsf{CNG\text{-}EDF}$ and $\mathsf{NP\text{-}CNG\text{-}EDF}$ reweighting algorithms.

Sporadic task systems. We denote the i^{th} task of a task system T as T_i (where tasks are ordered by some arbitrary method), and denote the j^{th} job of the task T_i as T_i^j (where jobs are ordered by the sequence in which they are invoked). A *sporadic task* is defined by an *execution cost*, denoted $\mathsf{e}(T_i)$, and *weight*, denoted $\mathsf{wt}(T_i)$, which specifies the fraction of a single processor it requires. (It is customary to define a sporadic task by its execution cost and the minimum separation time between its successive jobs—we define the latter in terms of weight and execution cost below.) As an example, Fig. 6(a) depicts a two-processor system scheduled via $\mathsf{CNG\text{-}EDF}$ with five tasks: T_1 with weight $2/7$ and execution cost 2; T_2 and T_3, each with an execution cost of 1 and weight at $3/7$; and T_4 and T_5, each with an execution cost of 3 and weight $3/7$. The first job of a task may be invoked or *released* at any time at or after time zero. The release time of job T_i^j is denoted $\mathsf{r}(T_i^j)$. Successive job releases of task T_i must be separated by at least $\mathsf{e}(T_i)/\mathsf{wt}(T_i)$ time units. For example, in Fig. 6(a), $\mathsf{r}(T_1^1) = 0$ and $\mathsf{r}(T_1^2) = 7$. The *absolute deadline* (or just *deadline*) of job T_i^j, denoted $\mathsf{d}(T_i^j)$, equals $\mathsf{r}(T_i^j) + \mathsf{e}(T_i)/\mathsf{wt}(T_i)$. For example, in Fig. 6(a), $\mathsf{d}(T_1^1) = 7$ and $\mathsf{d}(T_1^2) = 14$. We consider a sporadic task T_i to be *active* at time t if there exists a job T_i^j (called T_i*'s active job*) such that $t \in [\mathsf{r}(T_i^j), \mathsf{d}(T_i^j))$.

Dynamic sporadic task systems. A *dynamic sporadic task system* is an extension of a sporadic task system, where the weight of each task T_i is a function of time t and its execution cost can vary with each job T_i^j. We use $\mathsf{wt}(T_i, t)$ and $\mathsf{e}(T_i^j)$, respectively, to denote these two quantities. (For the remainder of the paper, whenever we refer to a "task" we are referring to a "dynamic sporadic task.") We use $\mathsf{wt}_{\mathsf{min}}(T_i)$ $(\mathsf{wt}_{\mathsf{max}}(T_i))$ to denote the *minimum* (*maximum*) *allowed weight for* T_i. As a shorthand, we use T_i:$[a, b]$ to denote a task T_i such that $\mathsf{wt}_{\mathsf{min}}(T_i) = a$ and $\mathsf{wt}_{\mathsf{max}}(T_i) = b$, and T_i:a to denote T_i:$[a, a]$. Furthermore, we use $\mathsf{e}_{\mathsf{max}}(T_i)$ to denote the maximal execution cost of any job of T_i.

For dynamic sporadic tasks, the *absolute deadline* of a job T_i^j is defined as

$$\mathsf{d}(T_i^j) = \mathsf{r}(T_i^j) + \mathsf{e}(T_i^j)/\mathsf{wt}(T_i, \mathsf{r}(T_i^j)).$$

In the absence of reweighting, consecutive job releases ($\mathsf{r}(T_i^j)$ and $\mathsf{r}(T_i^{j+1})$) of a task T_i must be separated by at least $\mathsf{e}(T_i^j)/\mathsf{wt}(T_i, \mathsf{r}(T_i^j))$ time units. For example, in Fig. 6(b), $\mathsf{r}(T_3^2) - \mathsf{r}(T_3^1) = 1/(3/7) = 7/3$, $\mathsf{r}(T_3^5) - \mathsf{r}(T_3^4) = 2/(4/7) = 7/2$ (T_3^4 is

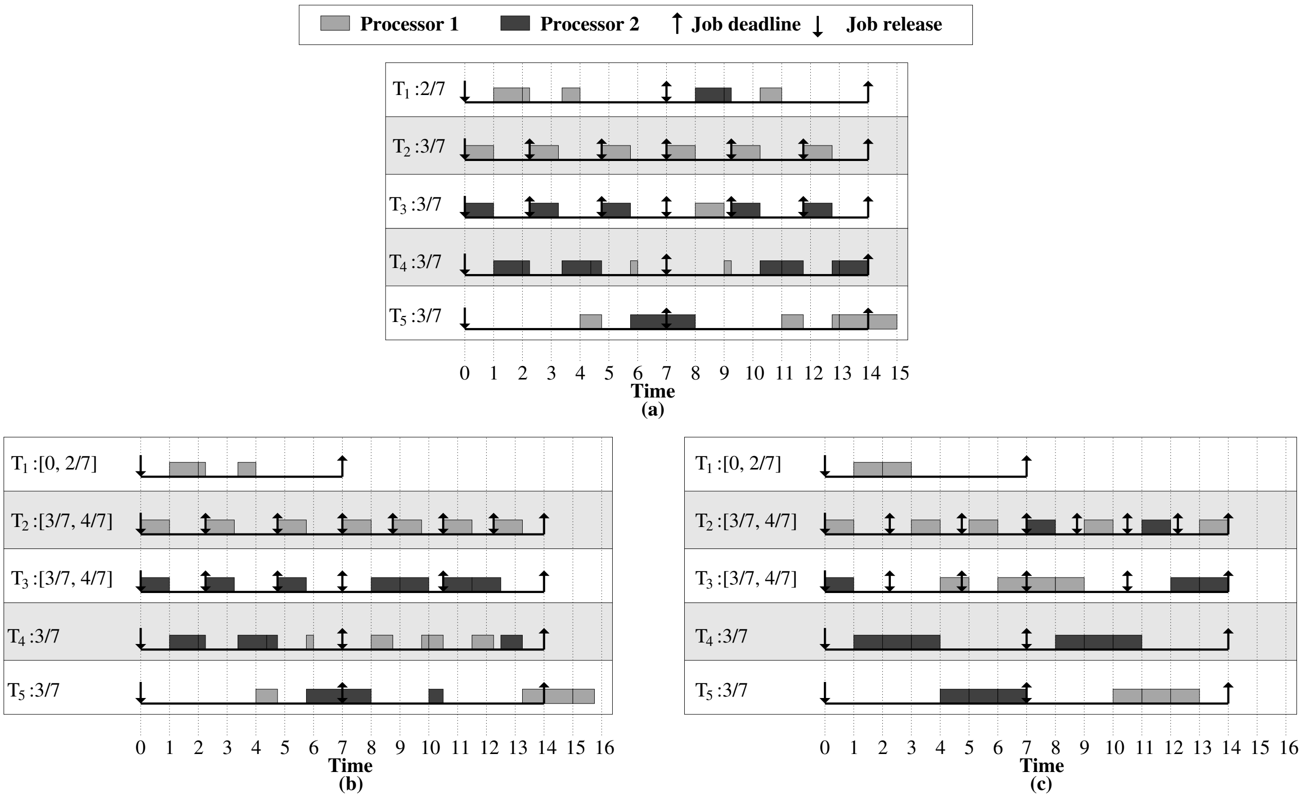


Figure 6: A two-processor system T with five tasks: T_1 with an execution cost of 2 and a weight of $2/7$; T_2 and T_3, each with an execution cost of 1 and a weight of $3/7$; and T_4 and T_5, each with an execution cost of 3 and a weight of $3/7$. Each job is *active* over an interval, denoted by the solid line, in which it should be scheduled. Inset **(a)** shows a CNG-EDF schedule without reweighting. In inset **(b)**, the system is scheduled by CNG-EDF, and at time 7 the following actions occur: T_1 leaves, T_2 and T_3 increase their weights to $4/7$, and T_3's execution cost increases to two. Inset **(c)** is the same system as in (b) except that it is scheduled by NP-CNG-EDF.

the first job released by T_3 after it changes weight and execution cost), and $\mathsf{d}(T_3^4) = 7 + 2/(4/7) = 10.5$.

A task T_i *changes weight* or *reweights* at time t if $\mathsf{wt}(T_i, t - \epsilon) \neq \mathsf{wt}(T_i, t)$ where $\epsilon \to 0^+$. We now explain some of the issues involved in processing such a reweighting event; in reading this overview, the reader may wish to refer to insets (b) and (c) in Fig. 6, which depict schedules for CNG-EDF and NP-CNG-EDF, respectively, in which some tasks are reweighted. (Other examples that illustrate specific details will be given later.) If a task T_i changes weight at a time t_c between the release and the deadline of some job T_i^j, then the following two actions *may* occur:

- The execution cost of T_i^j *may* be reduced to the amount of time for which T_i^j has executed prior to t_c, and the execution cost of T_i^{j+1} *may* be redefined to be the amount of time "lost" by reducing the execution cost of T_i^j.

- $\mathsf{r}(T_i^{j+1})$ *may* be redefined to be less than $\mathsf{r}(T_i^j) + \mathsf{e}(T_i^j)/\mathsf{wt}(T_i, \mathsf{r}(T_i^j))$. In this case, since $\mathsf{d}(T_i^j) = \mathsf{r}(T_i^j) + \mathsf{e}(T_i^j)/\mathsf{wt}(T_i, \mathsf{r}(T_i^j))$, jobs T_i^j and T_i^{j+1} will "overlap." (In the variant of the sporadic model defined earlier, every job's deadline is at or before its successor's release.)

The reweighting rules we present in Sec. 3 state under what conditions the above actions occur and by how much before $\mathsf{r}(T_i^j) + \mathsf{e}(T_i^j)/\mathsf{wt}(T_i, \mathsf{r}(T_i^j))$ the job T_i^{j+1} can be released. Since a reweighting event may cause a job's execution cost to decrease, we introduce the notion of a job T_i^j's *actual execution cost*, denoted $\mathsf{ae}(T_i^j)$, which represents the total amount of execution time that T_i^j will receive. (In Fig. 6, $\mathsf{ae}(T_i^j)$ equals $\mathsf{e}(T_i^j)$ for each job T_i^j, but later we will consider some examples where the two differ.)

When a task reweights, there can be a difference between when it "initiates" the change and when the change is "enacted." The time at which the change is *initiated* is a user-defined time; the time at which the change is *enacted* is dictated by a set of conditions discussed shortly. We use the *scheduling weight of a task T_i at time t*, denoted $\mathsf{swt}(T_i, t)$, to represent the "last enacted weight of T_i." Formally, $\mathsf{swt}(T_i, t)$ equals $\mathsf{wt}(T_i, u)$, where u is the last time at or before t that a weight change was enacted for T_i. It is important to note that, *henceforth, we compute task deadlines and releases using scheduling weights.* Hence we have the following formulas:

$$\begin{aligned} \mathsf{d}(T_i^j) &= \mathsf{r}(T_i^j) + \mathsf{e}(T_i^j)/\mathsf{swt}(T_i, \mathsf{r}(T_i^j)). \\ \mathsf{r}(T_i^{j+1}) &\geq \mathsf{d}(T_i^j). \end{aligned}$$

Notice that the second equation only applies in the absence of reweighting events, which may cause release times to be redefined.

Scheduling. Under both CNG-EDF and NP-CNG-EDF, "ready" jobs are prioritized by deadlines (which are based on scheduling weights), with earlier deadlines having higher priority. ("Ready" will be formally defined shortly.) Deadline ties are resolved arbitrarily. Under CNG-EDF, an arriving job with higher priority preempts the executing job with the lowest priority if no processor is available. The preempted job may later resume execution on a different processor. Under NP-CNG-EDF, the arriving job waits until some job completes execution and a processor becomes available. Thus, under NP-CNG-EDF, once scheduled, a job is guaranteed execution until completion without interruption.

Let $\mathcal{A}$ be an arbitrary scheduling algorithm, T be an arbitrary task system, and $\mathcal{S}$ denote an m-processor schedule of T generated by $\mathcal{A}$. Furthermore, let $\mathsf{A}(\mathcal{S}, T_i^j, t_1, t_2)$ denote the total time allocated to T_i^j in $\mathcal{S}$ in $[t_1, t_2)$. Similarly, we use $\mathsf{A}(\mathcal{S}, T_i, t_1, t_2)$ and $\mathsf{A}(\mathcal{S}, T, t_1, t_2)$, respectively, to denote the total time allocated to all jobs of T_i in $\mathcal{S}$ and all tasks of T in $\mathcal{S}$, over the interval $[t_1, t_2)$. We say that the value of $\mathsf{A}(\mathcal{S}, T_i^j, 0, t)$ is the amount that T_i^j has *executed by* t. For example in Fig. 6(b), $\mathsf{A}(\mathcal{S}, T_4^1, 0, 7) = 3$, $\mathsf{A}(\mathcal{S}, T_4^1, 1, 6) = 3$, and $\mathsf{A}(\mathcal{S}, T_4^1, 3, 7) = 5/3$.

Definition 1 (Halted). As discussed later, if a reweighting event in schedule $\mathcal{S}$ occurs at time t, then it is possible that some job T_i^j is *halted* at t. In this case, $\mathsf{ae}(T_i^j)$ is set to $\mathsf{A}(\mathcal{S}, T_i^j, 0, t)$. (An example of a halted task is given in Fig. 7(a), discussed shortly, in which T_4^1 halts when it is reweighted at time 2.)

Definition 2 (Completed). A job T_i^j is said to have *completed by time t in $\mathcal{S}$* iff T_i^j has executed for $\mathsf{e}(T_i^j)$ by t in $\mathcal{S}$, or T_i^j has halted by time t. A task T_i is said to have *completed at time t in $\mathcal{S}$* if at time t every job of T_i that has been released by t has completed.

Definition 3 (Active and Inactive). A job T_i^j is *active* at time t iff $\mathsf{r}(T_i^j) \le t < \mathsf{d}(T_i^j)$, $t < \mathsf{r}(T_i^{j+1})$ (if T_i^{j+1} exists) and no reweighting event before t in $\mathcal{S}$ has caused T_i^j to become *inactive* (for example, as a result of halting T_i^j). For example, in Fig. 7(c) (described in detail shortly), T_4^1 is active over the range $[0, 3)$. In this example, T_4^1 becomes inactive at 3 since $\mathsf{r}(T_4^2) = 3$.

Definition 4 (Pending and Ready). A job T_i^j is said to be *pending at time t in $\mathcal{S}$* if $t \ge \mathsf{r}(T_i^j)$ and T_i^j is not complete by t in $\mathcal{S}$. For example, in Fig. 6(a), the job T_5^1 is pending over the range $[0, 8)$. Note that a job can be pending, but not active, if it misses its deadline. A pending job T_i^j is said to be *ready at time t in $\mathcal{S}$* if all prior jobs of task T_i have completed by t. For example, in Fig. 6(a), the job T_5^2 is ready over the range $[8, 15)$. A job T_i^j can be pending but not ready if T_i^{j-1} has not completed by $\mathsf{r}(T_i^j)$.

3 Task Reweighting

We now introduce two new reweighting rules that are CNG-EDF extensions of the PD^2-OI reweighting rules presented by Block *et al.* previously [3]. As mentioned before, these rules work by modifying future job release times and execution costs. At the end of this section, we discuss how to adjust these rules for NP-CNG-EDF.

For simplicity, we assume that the actual execution cost for any job is equal to its specified execution cost, *unless* a task reweights when it has an active job. Then *and only then* can the actual execution cost of a job be less than its execution cost.[4] In this scenario, the actual execution cost of the job is determined by the rules we present shortly.

Let T be a task system in which some task T_i initiates a weight change to weight v at time t_c. Let w be the last scheduling weight of T_i before the change is initiated at t_c. Let $\mathcal{S}$ be the m-processor CNG-EDF schedule of T. Let T_i^j be last-released job of T_i before t_c. If T_i^j does not exist or T_i^j is inactive at t_c before the reweighting event is initiated (*i.e.*, $t_c \ge \mathsf{d}(T_i^j)$), then the weight change is immediately enacted, and future jobs of T_i are released with the new weight. In the following rules, we

[4]Since reweighting events may modify the actual execution cost of a job, removing this assumption would entail having notation to distinguish between redefined execution costs as the result of reweighting and redefined execution costs that occur due to a task executing for less than its specified execution cost.

consider the remaining possibility, *i.e.*, T_i^j exists and is active at t_c. (Notice that if $t_c = \mathsf{d}(T_i^k) = \mathsf{r}(T_i^{k+1})$, then T_i^k is the last-released job of T_i before t_c, and it is not active at t_c. Therefore, the change is immediately enacted and T_i^{k+1} is released with the new weight.)

Let $\mathsf{rem}(T_i^j, t_c) = \mathsf{e}(T_i^j) - \mathsf{A}(\mathcal{S}, T_i^j, 0, t)$. Note that $\mathsf{rem}(T_i^j, t_c)$ denotes the actual remaining computation in T_i's current job. Let NxtEx equal $\mathsf{rem}(T_i^j, t_c)$, if $\mathsf{rem}(T_i^j, t_c) > 0$; otherwise, if $\mathsf{rem}(T_i^j, t_c) = 0$, then let NxtEx equal the value of $\mathsf{e}(T_i^{j+1})$ had the weight-change event not occurred. Since NxtEx is only used to determine the size of the next job released, it can be calculated at time $\mathsf{r}(T_i^{j+1})$. (Notice that, if T_i has no next job T_i^{j+1} to release at the time specified in the rules below, then $\mathsf{NxtEx} = 0$. In this case, the rules are applied as stated, except that T_i^{j+1} is not released.)

The $\mathsf{SW\text{-}NC}$ scheduling algorithm and deviance. Recall from the introduction, that the choice of which reweighting rule to apply depends on whether the task is over- or under-allocated relative to the "ideal" schedule. The *non-clairvoyant scheduling-weight processor-sharing* ($\mathsf{SW\text{-}NC}$) scheduling algorithm is an ideal scheduling algorithm that is used for determining which rule to apply. (We introduce a clairvoyant scheduling-weight processor-sharing algorithm in Sec. 4.) Under the $\mathsf{SW\text{-}NC}$ scheduling algorithm, at each instant t, each active job T_i^j in T is allocated a share equal to its scheduling weight $\mathsf{swt}(T_i, t)$. Hence, if a job T_i^j is active over the range $[t_1, t_2)$, then over this range T_i^j is allocated $\int_{t_1}^{t_2} \mathsf{swt}(T_i, u)du$. (If a job is inactive, then it receives no allocations in $\mathsf{SW\text{-}NC}$.) Throughout this paper we use $\mathcal{N}$ to denote the $\mathsf{SW\text{-}NC}$ schedule of a task system T. For example, in Fig. 7(c) (described shortly), notice that while T_4's scheduling weight is constant at $1/6$ over the range $[0, 2)$, $\mathsf{A}(\mathcal{N}, T_4^1, 0, 2) = \int_0^2 \mathsf{swt}(T_4, u)du = 2/6$; however, when T_i's scheduling weight increases to $4/6$ at time 3, we have $\mathsf{A}(\mathcal{N}, T_4^1, 0, 3) = \int_0^3 \mathsf{swt}(T_4, u)du = 6/6$.

The *deviance of job T_i^j of task T_i at time t* is defined as $\mathsf{dev}(T_i^j, t) = \mathsf{A}(\mathcal{N}, T_i^j, 0, t) - \mathsf{A}(\mathcal{S}, T_i^j, 0, t)$. The deviance of a job T_i^j represents the difference between the actual and $\mathsf{SW\text{-}NC}$ allocations of T_i^j up to time t. If the deviance is negative, then the job has received a greater allocation in the actual schedule, and if the deviance is positive, then the job has received a greater allocation in the $\mathsf{SW\text{-}NC}$ schedule. The choice of which rule to apply depends on whether deviance is positive or negative. If positive, then we say that T_i is *positive-changeable at time t_c from weight w to v*; otherwise T_i is *negative-changeable at time t_c from weight w to v*. Because T_i initiates its weight change at time t_c, $\mathsf{wt}(T_i, t_c) = v$ holds; however, T_i'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_i, then the previous event is *canceled*, *i.e.*, treated as if it had not occurred. As discussed later, any "error" associated with canceling a reweighting event like this is accounted for when determining drift.

Rule P: If T_i is positive-changeable at time t_c from weight w to v, then one of two actions is taken: **(i)** if $\mathsf{d}(T_i^j) - t_c > \mathsf{rem}(T_i^j, t_c)/v$, then immediately, T_i^j is halted, the weight change is enacted, a new job of size NxtEx is released (if $\mathsf{NxtEx} > 0$), and T_i^j becomes inactive; **(ii)** otherwise, at time $\mathsf{d}(T_i^j)$, the weight change is enacted, *i.e.*, the scheduling weight of T_i does not change until the end of its current job.

Rule N: If T_i is negative-changeable at time t_c from weight w to v, then one of two actions is taken: **(i)** if $v > w$, then immediately, T_i^j is halted and its weight change is enacted, and at time t_r, a new job of size NxtEx is released (if $\mathsf{NxtEx} > 0$) and T_i^j becomes inactive, where t_r is the smallest time at or after t_c such that $\mathsf{dev}(T_i^j, t_r) = 0$ holds; **(ii)** otherwise, at time t_e, the weight change is enacted, a new job of size NxtEx is released (if $\mathsf{NxtEx} > 0$), and T_i^j becomes inactive, where $t_e = \min(t_r, \mathsf{d}(T_i^j))$, and t_r is smallest time at or after t_c such that $\mathsf{dev}(T_i^j, t_r) = 0$ holds.

Intuitively, Rule P changes a task's weight by halting its current job and issuing a new job of size NxtEx with the new weight if doing so would improve its deadline. A (one-processor) example of a positive-changeable task that changes its weight via Case (i) of Rule P is given in Fig. 7(a). (We discuss the terms "drift," "IDEAL allocations," and "SW allocations" in Sec. 4.) The depicted example consists of a task system T with four tasks as defined in the figure's caption. Since T_2, T_3, and T_4 have the same initial deadline, we have arbitrarily chosen T_4 to have the lowest priority. In inset (a), T_4 is positive-changeable since at time 2 it has not yet been scheduled. Note that halting T_4's current job and issuing a new job of size one improves T_4's scheduling priority, *i.e.*, $\mathsf{d}(T_4^1) = 6 > \frac{7}{2} = \mathsf{d}(T_4^2)$. Notice also that the second job of T_4 is issued 6/4 quanta after time 2. This spacing is in keeping with a new job of weight 4/6 issued at time 2. A one-processor example of a positive-changeable task that changes its weight via Case (ii) of Rule P is given in Fig. 7(b). The depicted example consists of a task system T with three tasks, T_1 with a weight of $1/3$, T_2 with a weight of $1/4$, and T_3 with an initial weight of $1/4$ that increases to $1/3$ at time 2. Notice that at time 2, T_3^1 has positive deviance since it has not yet been scheduled. Thus, when it changes weight it does so by Rule P. Notice also that if T_3^1 halted at time 2 and released a new job of weight $1/3$, the deadline of this new job would equal time 5 (since $5 = 2 + 1/(1/3)$). Thus, if we were to enact the change via Case (i) of Rule P, then we would in effect be *increasing* the deadline of the first scheduled job of T_3, even though the weight of the task increased. Therefore, we enact the weight change via Case (ii) of Rule P, which delays enacting the weight change until the deadline of T_3^1.

Rule N changes the weight of a task by one of two approaches: **(i)** if a task *increases* its weight, then Rule N causes the release time of its next job to be adjusted so that it is commensurate with the new weight; **(ii)** if a task *decreases* its weight, then Rule N causes the next job to be issued with a deadline that is commensurate with the new weight at the end of the current job.

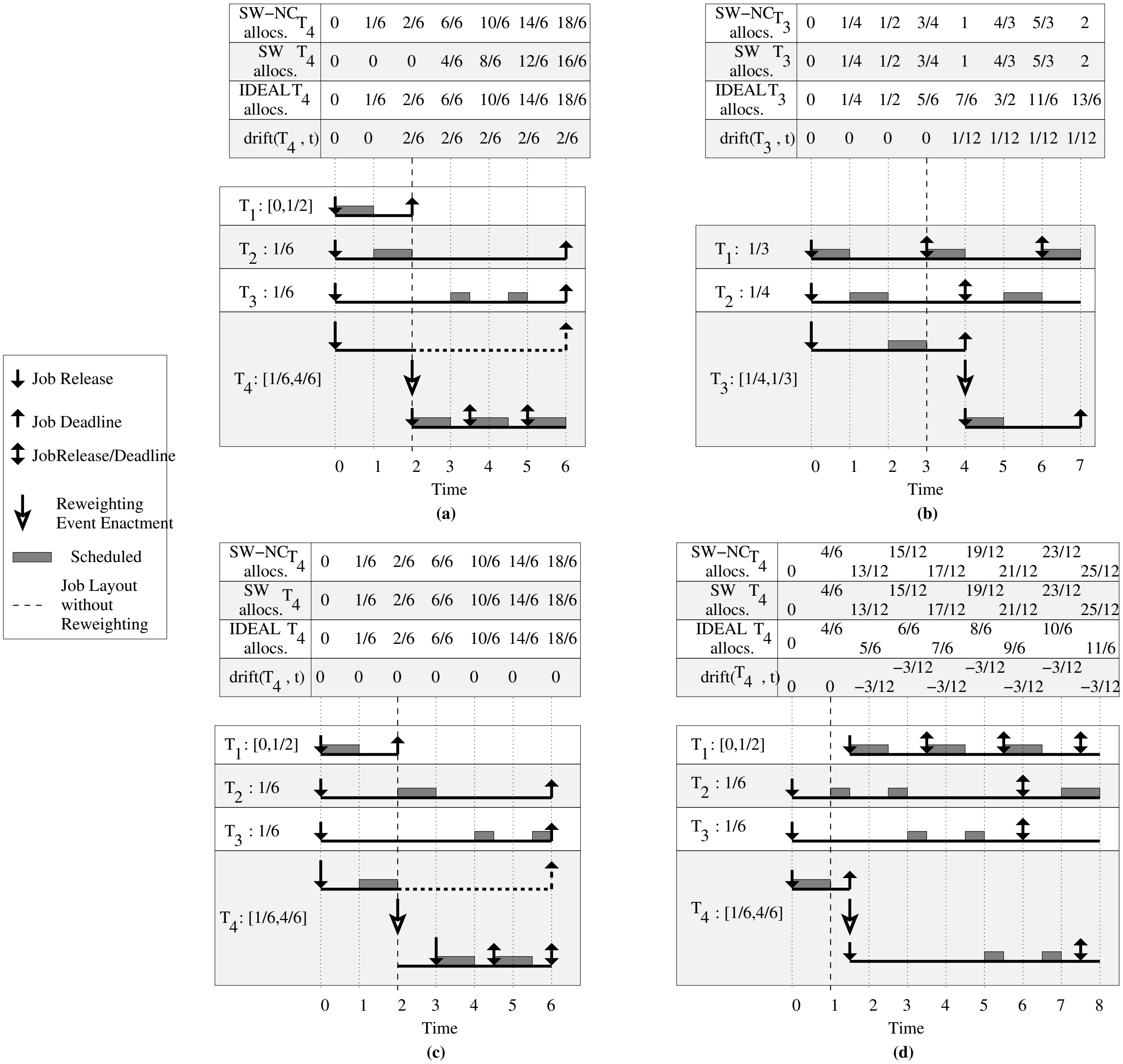


Figure 7: The CNG-EDF schedule for several one-processor systems. **(a)** A system consisting of four tasks (where the execution cost of each job is one): T_1:$[0, 1/2]$, which leaves at time 2, T_2:1/6, T_3:1/6, and T_4, which has an initial weight of $1/6$ that increases to $4/6$ at time 2. In this system, T_4 initially has the lowest scheduling priority. Since T_4 is not scheduled by time 2, it has positive deviance and changes its weight via Rule P, causing T_4^1 to be halted, T_4^2 to be released at time 2 with a deadline of 7/2, and T_4's drift to become 2/6. **(b)** A system consisting of three tasks (where the execution cost of each job is one): T_1 with a weight of $1/3$, T_2 with a weight of $1/4$, and T_3 with an initial weight of $1/4$ that increases to $1/3$ at time 2. Since T_1^1 has not been scheduled by time 2 its deviance is positive; furthermore, since $\mathsf{d}(T_3^1) - 2 < \mathsf{rem}(T_3^1, 2)/(1/3)$, T_1 enacts its weight change via Case (ii) of Rule P. **(c)** The same scenario as in (a) except that T_4 has higher priority than both T_2 and T_3. Since T_4 has been scheduled by time 2, it has negative deviance and changes its weight via Rule N(i), causing its next job to have a release time of 3 while maintaining a drift of zero. **(d)** T_1 joins the system at time 6/4 and T_4 has an initial weight of 4/6 that decreases to 1/6 at time 1. Since T_4 has negative deviance at time 1, its weight is changed via Rule N(ii), causing T_4's next job to have a deadline of 15/2 and T_4 to have a drift of $-3/12$. In all insets, the drift and allocations in the SW-NC, IDEAL, and SW schedules for either T_3 or T_4 are labeled as a function of time across the top.

A (one-processor) example of a negative-changeable task that increases its weight is given in Fig. 7(c). The depicted example consists of the same tasks as in (a), except that we have chosen T_4 to have the highest priority. Notice that the second job of T_4 is issued at time 3, which is the time such that $\mathsf{dev}(T_4, 3) = \int_0^3 \mathsf{swt}(T_i, u)du - \mathsf{A}(\mathcal{S}, T_4, 0, 3) = 1 - 1 = 0$. Recall that the deadline (release time) of the j^{th} ($(j+1)^{th}$) job of a task T_i is given by $\mathsf{r}(T_i^j) + \mathsf{e}(T_i^j)/(\mathsf{swt}(T_i, \mathsf{r}(T_i^j)))$. Hence, if a task T_i of weight v were to issue a job of size $y = \mathsf{A}(\mathcal{S}, T_i^j, 0, t_c) - \mathsf{dev}(T_i^j, t)$ at time t_c, then the release time of its next job would be $t_c + y/v$. A (one-processor) example of a negative-changeable task that decreases its weight is given in Fig. 7(d). The depicted example consists of the same four tasks except that T_4 has an initial weight of 4/6 and decreases its weight at time 1, and T_1 joins the system as soon as T_4's weight change is enacted.

Notice that if T_i initiates a weight change at time t_c while some job T_i^k of T_i (not necessarily its last-released job) has missed its deadline, then the Rules P and N specify that one of two actions is taken. If no job of T_i is active at t_c, then the weight change is enacted immediately. If there is a job T_i^j that is active at t_c, then since T_i^j has not been scheduled (because the earlier job T_i^k has missed its deadline), it follows that T_i is positive-changeable, and thus the weight change is enacted via Rule P (which may cause T_i^j *but not* T_i^k to halt). Notice that T_i^k is unaffected in both cases.

It is important to remember that when the rules P and N halt a job, they do not abandon the process that the job was working on. But rather, these rules split the process into two jobs. Since these rules change the ordering of a task in the priority queues that determine scheduling, the time complexity for reweighting one task is $O(logN)$, where N is the number of tasks in the system.

Canceled reweighting events. We now introduce a property about the relationship between the initiation and enactment of a reweighting event in the case that some such events are canceled due to later reweighting events. Notice that, once a task T_i initiates a weight change at t_c, this weight change is eventually either canceled by another weight change or enacted. Further, Rules P and N enact any non-canceled reweighting event no later than the deadline of the last-released job T_i^j of T_i at t_c (if it exists and if $t_c \leq \mathsf{d}(T_i^j)$). For example, consider the one-processor system in Fig. 8. The depicted system consists of three tasks, each with an execution cost of 2: T_1 has an initial weight of $1/3$ that changes to $1/10$ at time 3 and changes again to $1/4$ at time 5; and T_2 and T_3 both have a weight of $1/3$. Since, in this example, T_1^1 has negative deviance at time 3 and is requesting a weight decrease, its weight change is enacted by Case (ii) of Rule N, which specifies that its change is enacted at time 6. Furthermore, when T_1 initiates the second change at time 5, the weight-change event initiated at time 3 is canceled, since it has not yet been enacted. Even though at time 5 the weight change initiated at time 3 is canceled, a new reweighting event is initiated, which is

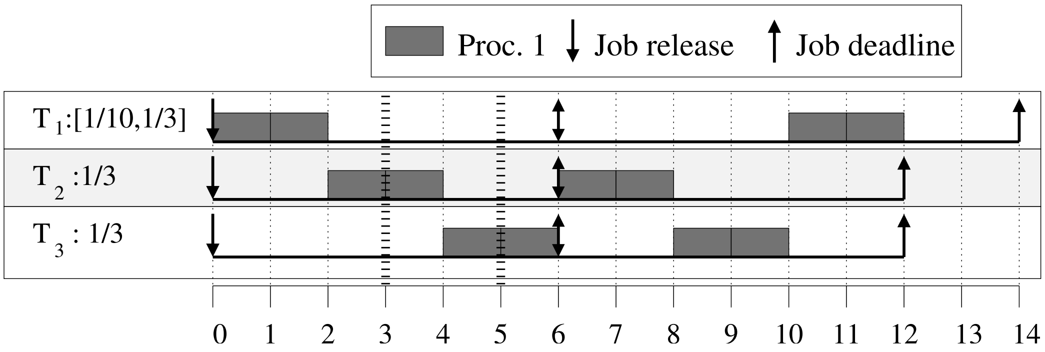


Figure 8: A one-processor system with three tasks: T_1 with an execution cost of 2 and an initial weight of $1/3$ that changes to $1/10$ at time 3 and then to $1/4$ at time 5; T_2 and T_3, each with an execution cost of 2 and a weight of $1/3$. Notice that, because the change initiated at time 3 is via Rule N (ii), the change is not immediately enacted, and as a result when the change initiated at time 5 is processed, the change initiated at time 3 is canceled.

in turn enacted at time 6. Thus, we can see that, once a reweighting event has been initiated during an active job, some weight change will be enacted by the earlier of the deadline of that job or when the job becomes inactive (which may be earlier, by Rules P and N). Property (X) formalizes this idea.

(X) If a task T_i initiates a weight change at time t_c and the job T_i^j is active at t_c, then some weight change is enacted according to Rule P or N by either $\mathsf{d}(T_i^j)$ or when T_i^j becomes inactive, whichever is first.

Modifications for NP-CNG-EDF. In order to adapt the Rules P and N to work for NP-CNG-EDF, the only modification we need to make is when these rules are *initiated*. If a task with an active job reweights *before or after* that job has been scheduled, then Rules P and N are initiated as before. (Note that if the active job *has not* been scheduled, then its deviance is positive, and if the active job *has* been scheduled, then its deviance is negative.) However, if a task changes its weight while the active job T_i^j is executing, then the initiation of the weight change is delayed *until T_i^j has completed* or *T_i^j is no longer active*, whichever is first. Note that, if a task T_i changes its weight from u to v at time t_c in NP-CNG-EDF, then $\mathsf{wt}(T_i, t_c) = v$ holds, regardless of whether the initiation of Rule P or N must be delayed. For example, consider the one-processor system depicted in Fig. 9 consisting of three tasks: T_1 with an execution cost of 1 and a weight of $1/2$ that leaves at time 2; T_2 with an execution cost of 1 and a weight of $1/6$; and T_3 with an execution cost of 2 and an initial weight of $1/3$ that increases to $4/6$ at time 2. Fig. 9(a) depicts a schedule where T_2 initially has higher priority than T_3, and as a result, when the change is initiated at time 2, it can be immediately enacted by Case (i) of Rule P. Fig. 9(b) depicts a schedule where T_2 initially has lower priority than T_3. Since T_3^1 is scheduled at time 2, the initiation of the weight change must be delayed until time 3, at which point, the change is enacted via Case (i) of Rule N.

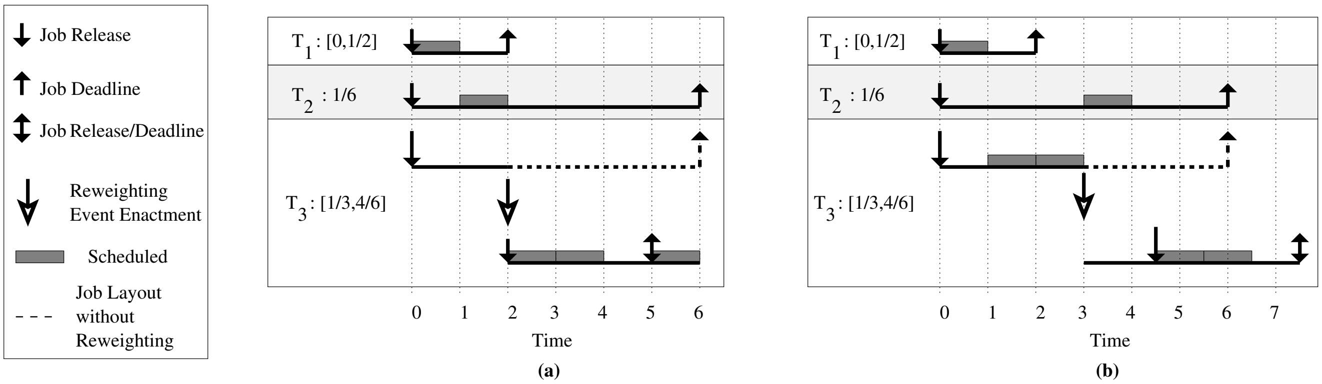


Figure 9: NP-CNG-EDF schedules for a one-processor system. **(a)** A system consisting of three tasks, T_1:$[0, 1/2]$ with an execution cost of 1 that leaves at time 2, T_2:$1/6$, and T_3 with an execution cost of 2 and an initial weight of $1/3$ that increases to $4/6$ at time 2. In this system, T_3 has the lowest scheduling priority. Since T_3 is not scheduled by time 2, it has positive deviance and changes its weight via Rule P, causing T_3^1 to be halted, T_3^2 to be released at time 2 with a deadline of 5. **(b)** The same scenario as in (a) except that T_3 has higher priority than T_2. Since task T_3 is being scheduled at time 2, and the system is schedule by NP-CNG-EDF, the initiation of the reweighting event is delayed until T_3 stops executing at time 3. Since T_3^1 is complete by time 2, it has negative deviance and changes its weight via Rule N, causing its next job to have a release time of $9/2$.

Scheme	Summary
SW-NC	While a task is active, this algorithm allocates the task its *scheduling weight* at each instant.
SW	While a task is active *and* the allocation to the active job is less than the actual execution cost, this algorithm allocates the task its *scheduling weight* at each instant.
IDEAL	While a task is active, this algorithm allocates the task its *weight* at each instant.

Table 3: Brief description of the different "ideal" scheduling algorithms considered in this paper.

4 Tardiness and Drift Bounds

In this section, we formally present and prove tardiness and drift bounds for the CNG-EDF algorithm. Because any set of reweighting rules will cause the "actual" schedule to deviate from the "ideal" schedule, the tardiness bounds reflect CNG-EDF's accuracy at *scheduling* the job set it created. The drift bounds, on the other-hand, reflect CNG-EDF's accuracy at creating a job set that mimics the "ideal" task system, where weight changes can always be initiated and enacted instantaneously. To this end, we introduce two additional theoretical scheduling algorithms: the *clairvoyant scheduling-weight processor-sharing* (SW) scheduling algorithm and the *ideal processor-sharing* (IDEAL) scheduling algorithm. Both algorithms have the ability to preempt and swap tasks at arbitrarily small intervals. SW-NC and SW allocate each task a share equal to its *scheduling weight*; however, SW *will not allocate* capacity to a task if its active job has received an allocation equal to its actual execution cost. IDEAL, on the other hand, allocates each task a share equal to its *weight* at each instant; and, unlike SW, IDEAL *will not stop allocating* capacity to a task while the task has an active job. We provide below a more in-depth explanation of these two algorithms. (For a quick reference, a brief description of all three algorithms appears in Table 3.)

The SW scheduling algorithm. SW is an ideal algorithm for scheduling the job set created by CNG-EDF that is used as a reference for assessing properties of actual schedules created by CNG-EDF. Under SW, at each instant t, each job of each task T_i that is both active and non-complete is allocated a fraction of a processor equal to $\mathsf{swt}(T_i, t)$. Furthermore, we consider SW to be "clairvoyant" in the sense that SW can use the value of $\mathsf{ae}(T_i^j)$ to determine if T_i^j has completed before it halts. More specifically, for any schedule $\mathcal{S}W$ under SW of any task system T, we say that T_i^j has *completed by time t in $\mathcal{S}W$* iff T_i^j has executed for $\mathsf{ae}(T_i^j)$ by t. Thus, the difference between SW-NC and SW is that a job in SW-NC will not stop receiving allocations as long as its active, whereas a job in SW will stop as soon as it receives its actual execution cost. For example, consider the one-processor task system T depicted in Fig. 10. Inset (a) depicts a CNG-EDF schedule for T, inset (b) depicts T's SW-NC schedule, and inset (c) depicts T's SW schedule. Notice that, in the SW schedule, T_1 does not receive any allocations over the interval $[3, 6)$. This is because at time 3 the total allocation to T_1^1 in the SW schedule equals $\mathsf{ae}(T_1^1) = 1$, hence, T_1^1 is complete at time 3. However, at time 6, T_1^2 is released, and therefore T_1 has an incomplete job with a scheduling weight of 1/2. Hence, T_1 begins to receive an allocation equal to its scheduling weight, which is now 1/2. This differs from the SW-NC, where T_1 receives allocations over the range $[3, 6)$. Notice that since SW does not allocate capacity to a task after it has executed for its actual execution cost, the SW schedule of a task T is a more accurate representation of the amount of allocation that T receives in $\mathcal{S}$ than the SW-NC schedule of T. Note that we assume that every job release, deadline, execution cost, and actual execution cost for a SW schedule to be the same as that in the corresponding CNG-EDF schedule.

The IDEAL scheduling algorithm. Under the *ideal processor sharing* (IDEAL) scheduling algorithm, at each instant t, each task T_i in T with an active job at t is allocated a share equal to its weight $\mathsf{wt}(T_i, t)$. Hence, if $\mathcal{I}$ is the IDEAL schedule of T and T_i is active over the interval $[t_1, t_2)$, then over $[t_1, t_2)$, the task T_i is allocated $\mathsf{A}(\mathcal{I}, T_i^j, t_1, t_2) = \int_{t_1}^{t_2} \mathsf{wt}(T_i, u)du$ time. As mentioned earlier, the IDEAL algorithm is similar to SW-NC, with one major exception: under IDEAL, each task receives an allocation equal to its *weight*, whereas under SW-NC, each task receives an allocation equal to its *scheduling weight*. For example, consider the IDEAL schedule of the task system T depicted in Fig. 7(b). Notice that, over the range $[3, 4)$, the task T_3 receives allocations equal to its weight at every instant, *i.e.*, $1/3$. Compare this to the SW-NC schedule, in which T_3 receives an allocation equal to its scheduling weight at every instant, *i.e.*, $1/4$. To see the difference between the IDEAL and SW schedule consider the IDEAL schedule of the task system T depicted in Fig. 10(c). Notice that, over the range $[3, 6)$, task T_1 receives allocations equal to its weight at every instant. Compare this to the SW schedule (inset (b)), in which T_1 receives *no* allocations over the range $[3, 6)$.

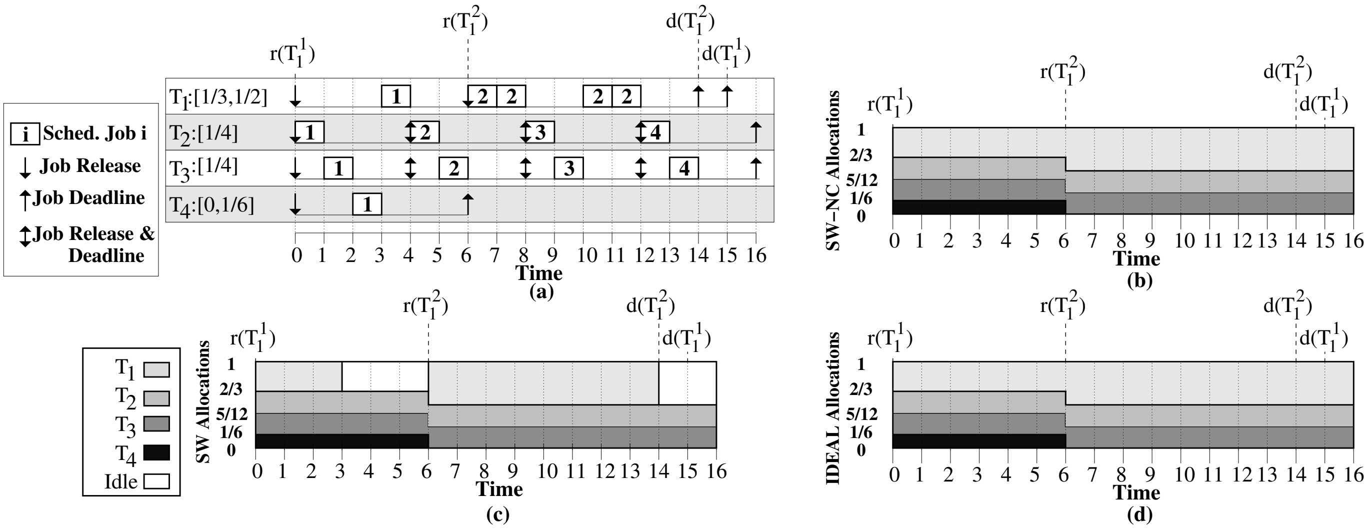


Figure 10: A one-processor task system T with four tasks: T_1 with $\mathsf{e}(T_1^1) = 5$ and an initial weight of 1/3 that increases to 1/2 via Case (i) of rule P at time 6, and as a result, $\mathsf{ae}(T_1^1) = 1$, $\mathsf{e}(T_1^2) = 4$, $\mathsf{r}(T_1^2) = 6$ and $\mathsf{d}(T_1^2) = 14$; T_2 with a constant weight of 1/4 and a constant execution cost of 1; T_3 with a constant weight of 1/4 and a constant execution cost of 1; and T_4 with an initial weight of 1/6 that decreases to 0 at time 6 and $\mathsf{e}(T_4^1) = 1$. Inset **(a)** depicts the CNG-EDF schedule of T. The number in each box denotes which job is scheduled, *e.g.*, over the range $[0, 1)$, T_2^1 is executing, and over the range $[4, 5)$, T_2^2 is executing. Inset **(b)** depicts the SW-NC schedule of T. Inset **(c)** depicts the SW schedule of T. Note that since $\mathsf{ae}(T_1^1) = 1$ at time 3, T_1^1 is complete in SW, *i.e.*, T_1 receives no allocations under SW over the range $[3, 6)$. Inset **(d)** depicts the IDEAL schedule of T. Note that the IDEAL allocation to any task T_i equals $\int_{t_1}^{t_2} \mathsf{wt}(T_i, u)du$ if T_i is active over the range $[t_1, t_2)$. For insets (b), (c) and (d), the releases and deadlines of the jobs T_1^1 and T_1^2 are depicted.

4.1 Tardiness Bounds

Instead of deriving tardiness bounds under CNG-EDF or NP-CNG-EDF from scratch (which would be quite tedious), we instead leverage the results reported by Devi and Anderson in [8] (the journalized version of [7]) concerning tardiness bounds that can be guaranteed under EDF and non-preemptive EDF (NP-EDF). In [8], apart from deriving tardiness bounds under EDF and NP-EDF for sporadic task systems, Devi and Anderson also proposed an extension to the sporadic task model, referred to as the *extended sporadic task model*, and determined tardiness bounds that can be guaranteed to task systems that conform to the extended sporadic task model. We will show that any dynamic task system can be modeled as an extended sporadic task system; hence, tardiness bounds derived for extended sporadic task systems can be applied to dynamic task systems as well. We begin by describing the extended sporadic task model.

The extended sporadic task model. In the conventional sporadic task model, the number of tasks in a task system is fixed, and the sum of the weights of all its tasks is assumed to be at most m (the number of processors). On the other hand, in the extended model, the number of tasks associated with a task system is allowed to vary and the total weight of all tasks is allowed to exceed m. (The number of tasks could potentially be infinite.) Further, each task is assigned a static weight, and all jobs (except possibly the final job) of a task have equal execution costs. However, to prevent overload, at any given time, only a subset of tasks whose total weight is at most m is allowed to be *effective*,[5] *i.e.*, is allowed to release jobs. Additionally, the final job of a task can *stop*[6] at some time t_s before its deadline, provided the allocation that the job receives in the actual schedule is at most the allocation it receives up to t_s in the SW-NC schedule, *i.e.*, the last job has non-negative deviance, and the job is not executing in a non-preemptive segment at t_s. When a job stops, its execution cost is altered to equal the amount of time that the job actually executed for in the actual schedule up to time t_s. Hence, the stopping job receives no allocation in the SW-NC schedule after t_s, even if its deadline is after t_s. Thus, at any time t, each task T_i can be in one of the following states.

- *Effective*, if the first job of T_i is released at or before t, the deadline of its final job is after t, and its final job has not stopped at or before t. A task whose final job has its deadline at or before t is not considered effective at t even if the final job is pending at t.
- *Ineffective*, if the release time of the first job of T_i is after t.
- *Terminated*, if the deadline of T_i's final job is before t.
- *Stopped*, if the final job has stopped but its deadline has not elapsed.

As can be easily seen, a task that is either ineffective or terminated at time t cannot have active or effective jobs at t. Also, although the deadline of a stopped job may be after the time at which it stops, the job receives no allocation in the SW-NC schedule after it stops.

In order to provide tardiness bounds for extended sporadic task systems, Devi and Anderson proposed partitioning the set of all tasks associated with a task system into N task classes such that the following hold: **(i)** effective intervals are disjoint for every two tasks in each class and **(ii)** tasks within a class are governed by precedence constraints, *i.e.*, the first job of a task cannot begin execution until all jobs of all tasks with earlier effective intervals in its class have completed execution. The second requirement implies that tasks that are not bound by precedence constraints should belong to different classes even if their effective intervals are disjoint. For example, consider the system given in Fig. 11(a), which consists of five tasks each with an execution cost of one: Z_1 with a weight of $1/2$ that leaves at time 2; Z_2 and Z_3, each with a weight of $1/6$; Z_4 with a weight of $1/6$ that stops at time 2; and Z_5 with a weight of $4/6$ that is in the same task class as Z_4. Notice that Z_4 can stop at time 2 because its actual allocation until then is no greater than its SW-NC allocation. Also note that, since Z_4 and Z_5 are in the same task class, only one of them can be effective at the same time.

[5] In [8], an effective task is referred to as an *active* task. We use this alternative term here to avoid conflicts in terminology.
[6] Here again, to avoid conflicting terminology, we differ from the term used in [8]. Stopping is referred to as *halting* in [8].

Let $T^{[\ell]}$ denote task class ℓ, and $\mathsf{e}_{\mathsf{max}}(T^{[\ell]})$ and $\mathsf{wt}_{\max}(T^{[\ell]})$, the maximum execution cost and weight, respectively, of any task in $T^{[\ell]}$. In [8], it is shown that the tardiness for any task of any task class $T^{[i]}$ of an extended sporadic task system T under global EDF is at most

$$\frac{\sum_{T^{[\ell]} \in \mathcal{E}_{\max}(T,\, m-1)} \mathsf{e}_{\mathsf{max}}(T^{[\ell]})}{m - \sum_{T^{[\ell]} \in \mathcal{X}_{\max}(T,\, m-2)} \mathsf{wt}_{\max}(T^{[\ell]})} + \mathsf{e}_{\mathsf{max}}(T^{[i]}), \tag{1}$$

and that under global NP-EDF is at most

$$\frac{\sum_{T^{[\ell]} \in \mathcal{E}_{\max}(T,\, m)} \mathsf{e}_{\mathsf{max}}(T^{[\ell]})}{m - \sum_{T^{[\ell]} \in \mathcal{X}_{\max}(T,\, m-1)} \mathsf{wt}_{\max}(T^{[\ell]})} + \mathsf{e}_{\mathsf{max}}(T^{[i]}), \tag{2}$$

where $\mathcal{E}_{\max}(T, k)$ and $\mathcal{X}_{\max}(T, k)$ are subsets of k task classes of T with the highest execution costs and weights, respectively, for any of their tasks (*i.e.*, with the highest values for $\mathsf{e}_{\mathsf{max}}(T^{[\ell]})$ and $\mathsf{wt}_{\max}(T^{[\ell]})$, respectively).

Relating extended sporadic task systems and dynamic task systems. We now show how a dynamic task system can be modeled as an extended sporadic task system. We initially assume that no task changes its weight by Case (i) of Rule N, *i.e.*, no negative-changeable task halts. Such weight changes are considered afterwards. We first show that each task of a dynamic sporadic task system can be modeled as a task class of an extended sporadic task system. For this, we decompose each dynamic task into disjoint "sub-tasks[7]," where a *sub-task* $\mathsf{sub}(T_i, j)$ of a dynamic task T_i is a "maximal" set of jobs with the following properties: **(i)** the jobs in $\mathsf{sub}(T_i, j)$ are consecutive jobs of T_i; **(ii)** each job is released between the same pair of two consecutive weight-change enactments for T_i; **(iii)** each job has the same execution cost; and **(iv)** no new job can be added to $\mathsf{sub}(T_i, j)$ without violating one or more of properties (i), (ii), and (iii), and in that sense, $\mathsf{sub}(T_i, j)$ is maximal. As an example, consider Fig. 11(c) and (d), which depict the same systems as in Fig. 7(a) and (c), respectively, with the sub-tasks marked.

Recall that in an extended sporadic task system, if a job T_i^j "stops" at time t_s, then $t_s < \mathsf{d}(T_i^j)$, T_i^j's deviance is non-negative, and that when a job stops its actual execution cost is set to the value that the job had executed for in the actual schedule up to time t_s. Since we are assuming that only positive-changeable tasks may halt, which by definition have an active job that has non-negative deviance, it is easy to see that for positive-changeable tasks a "stop" in the extended sporadic task system has the same effect as a "halt" in the dynamic sporadic task system. For example, the impact on the system when Z_4^1 stops in Fig. 11(a) is the same as when T_4^1 halts in Fig. 11(c).

By definition, all jobs of a sub-task have equal execution costs, and because all such jobs are released between two consecutive weight-change enactments, each sub-task has a static scheduling weight. Also, as explained above, halting is the same as

[7]This usage of the term "sub-task" should not be confused with that used in work on Pfair scheduling.

stopping for a positive-changeable task. Hence, if no task changes its weight via Case (i) of Rule N, then it follows that each sub-task in the dynamic sporadic task model corresponds to a task, with a static weight and execution cost, in the extended sporadic task model, and each task in a dynamic sporadic task model that consists of sub-tasks corresponds to a task class, composed of tasks with different weights or execution costs or both, of the extended sporadic task model. Also note that intervals within which sub-tasks of a dynamic task are effective are disjoint. For example, consider Fig. 11(a) and (c), which despite the notation change, have the identical schedules.

We now explain that a dynamic sporadic task can be modeled as an extended sporadic task even if tasks change their weight via Case (i) of Rule N. Before we continue, notice that the one difference between positive and negative-changeable tasks with respect to halting at time t_h is as follows: if T_i is positive-changeable at t_h and its job T_i^j is active at that time, then T_i^{j+1}, *i.e.*, the next job of T_i, may be released at t_h, whereas if T_i is negative-changeable, then T_i^{j+1} may not be released until time $t_r > t_h$, where t_r is the earliest time at which the allocations to T_i^j are equal in the actual schedule and under SW-NC, *i.e.*, the next time T_i^j's deviance is zero. Thus, if T_i is negative-changeable and halts at t_h, then T_i^j may be thought of as stopping at time t_r, where t_r is as defined above. However, notice that by Case (i) of Rule N, if T_i^j halts at time t_h, then T_i enacts a weight increase at time t_h. Thus, the time t_r is calculated using dynamic weights, which are not explicitly included in Devi and Anderson's extended sporadic model. For example, in Fig. 11(d), T_4^1 halts at time 2 and its successor is released at time 3, and in Fig. 11(b), which shows an extended sporadic system equivalent to Fig. 11(d), Z_4^1 does not stop until time 3. However, if we were using static weights, then Z_4^1 in Fig. 11(b) would not be able to stop until time 6.

Even though dynamic weights are not explicitly included in the extended sporadic model, the bounds in (1) and (2) still hold if the only time a task is allowed to change its weight is when its final job has finished executing and the weight change is an increase, which is exactly the scenario that arises when a task changes its weight via Case (i) of Rule N. The reason why (1) and (2) still hold in the presence of such weight changes is because the extended sporadic task model only requires that the total allocation in the SW-NC schedule to a stopping job T_i^j at the time it stops be at least the allocation T_i^j received in the actual schedule; increasing the rate at which the T_i^j is allocated time in the SW-NC schedule is not an issue as long as the total SW-NC allocation to all tasks that are effective is at most m at each instant. (Informally, this holds by the following reasoning. Increasing the allocation rate to a task T_i can only increase the scheduling priority of T_i. However, since the final job of T_i has already completed execution in the actual schedule when it halts, this increase in priority does not impact any job. In particular, it is not possible for another job to have a lower priority than the halting job T_i^j before the weight change and a higher priority after the weight change. Therefore, scheduling T_i^j (the halting job) in the past with a lower priority does not adversely impact

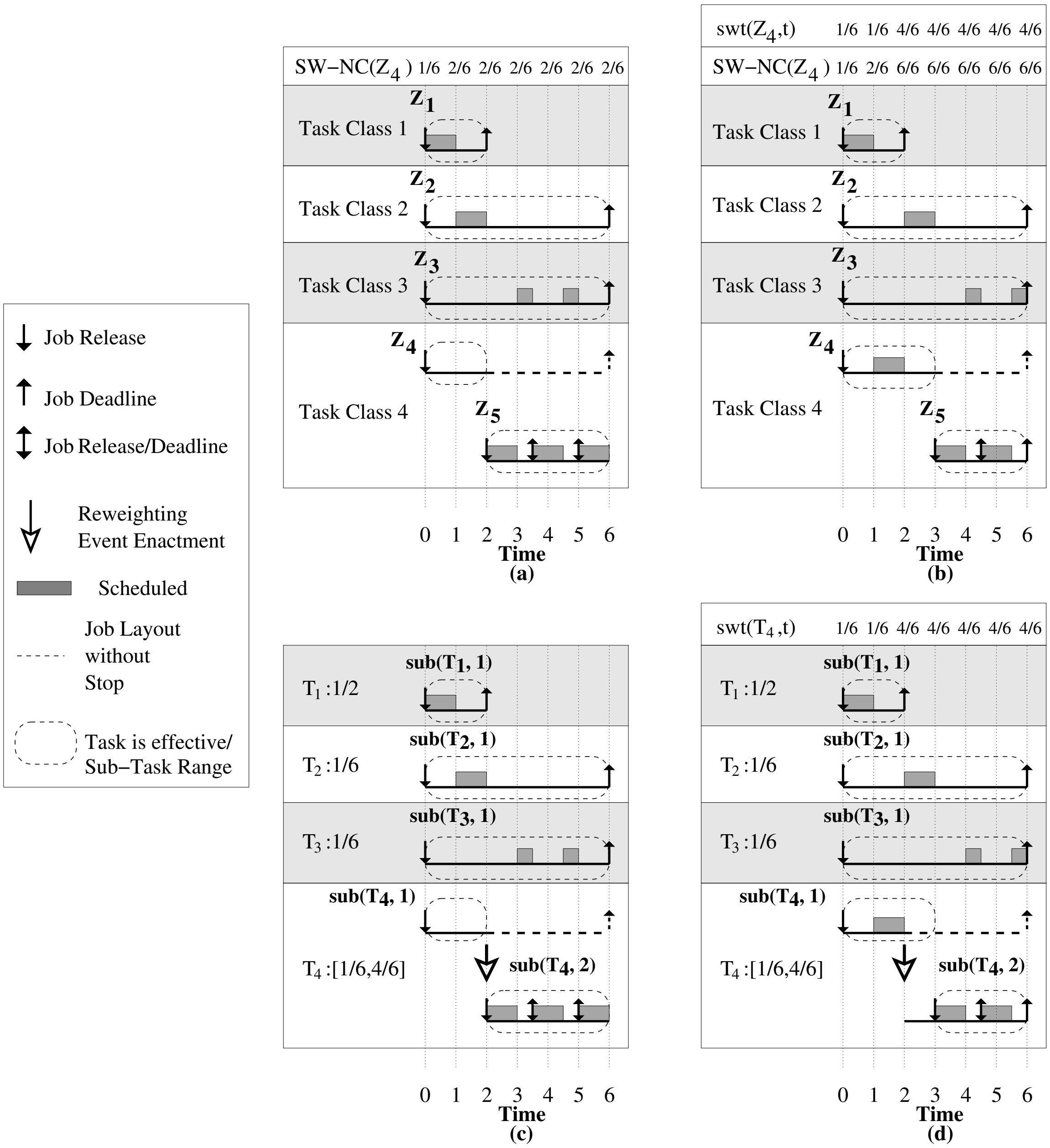


Figure 11: A one-processor system consisting of five tasks, each with an execution cost of one: Z_1 with weight $1/2$ that leaves at time 2; Z_2 and Z_3, each of which has a weight of $1/6$; Z_4 with a weight of $1/6$, which at time 2 stops in inset (a) and increases its weight to $4/6$ in inset (b); Z_5 with a weight of $4/6$ that is in the same task class as Z_4 and becomes effective as soon as Z_4 halts. **(a)** The tie among Z_2, Z_3, and Z_4 at time 1 is broken in favor of Z_2. As a result, at time 2, Z_4^1 has not received a greater allocation in the actual than in the SW-NC schedule, and therefore, Z_4^1 can (and in fact does) stop at time 2. **(b)** The tie among Z_2, Z_3, and Z_4 at time 1 is broken in favor of Z_4. As a result, at time 2, Z_4^1 has received a greater allocation in the actual than in the SW-NC schedule, and therefore, Z_4^1 must wait until time 3 before stopping. In both of these insets, the dashed rectangles with rounded corners denote when the task is effective. Also, Z_4's allocations under both SW and SW-NC are shown at the top of both insets. Insets **(c)** and **(d)** depcit the same systems as in Fig. 7(a) and (c), respectively. In both insets, each sub-task is denoted by a dashed rectangle with rounded corners.

how any other job was scheduled in the past. Since T_i^j has already completed execution before its deadline, T_i^j's tardiness is not impacted either.) Hence, if the task T_i enacts a weight change via Case (i) of Rule N at t_h and the system is not over-utilized after the change, no other job will be impacted. Thus, since a weight change is enacted at t_h regardless of whether T_i is positive- or negative-changeable, and T_i^{j+1} is released at or after t_h, in both the cases, T_i^j and T_i^{j+1} belong to different sub-tasks of T_i. Hence, the definition of a sub-task is unaltered even in the presence of negative-changeable jobs by Case (i) of Rule N, and the correspondence described earlier between a dynamic sporadic task and an extended sporadic task holds.

Thus, the tardiness bounds specified in (1) and (2) and that can be guaranteed to extended sporadic task systems are also applicable to dynamic sporadic task systems if task class $T^{[\ell]}$ is replaced by dynamic sporadic task T_z, and $\mathsf{e}_{\mathsf{max}}(T_z)$ and $\mathsf{wt}_{\max}(T_z)$ are taken as the maximum execution cost of any job of T_z and the maximum weight assigned to T_z at any time. (It should be noted that the tardiness bounds hold only if the sum of the weights of all tasks that are active at any instant is at most m.) Thus, we have the following theorem.

Theorem 1. *Let T be a dynamic task system, where for any $t \geq 0$, $\sum_{T_i \in T} \mathsf{swt}(T_i, t) \leq m$. Then, for any task T_i,* CNG-EDF *on m processors ensures a tardiness of at most*

$$\frac{\sum_{T_z \in \mathcal{E}_{\max}(T,\, m-1)} \mathsf{e}_{\mathsf{max}}(T_z)}{m - \sum_{T_z \in \mathcal{X}_{\max}(T,\, m-2)} \mathsf{wt}_{\mathsf{max}}(T_z)} + \mathsf{e}_{\mathsf{max}}(T_i),$$

and NP-CNG-EDF *on m processors ensures a tardiness of at most*

$$\frac{\sum_{T_z \in \mathcal{E}_{\max}(T,\, m)} \mathsf{e}_{\mathsf{max}}(T_z)}{m - \sum_{T_z \in \mathcal{X}_{\max}(T,\, m-1)} \mathsf{wt}_{\mathsf{max}}(T_z)} + \mathsf{e}_{\mathsf{max}}(T_i).$$

4.2 Drift

We now turn our attention to the issue of measuring "drift" under CNG-EDF. In order to measure the "drift" of a task system T, we compare the SW schedule of T to its corresponding IDEAL schedule.

For most real-time scheduling algorithms, the difference between the ideal and actual allocations a task receives lies within some bounded range centered at zero. For example, under a *uniprocessor* EDF schedule, the difference between the ideal and actual allocations for a task lies within $(-\mathsf{e}_{\mathsf{max}}(T_i), \mathsf{e}_{\mathsf{max}}(T_i))$. When a weight change occurs, the same bounds are maintained except that they may be centered at a different value. For example, in Fig. 7(a), the range is originally $(-1, 1)$, but after the reweighting event, it is $(-4/6, 8/6)$. This lost allocation is called *drift*. Given this loss (barring further reweighting events) T_i's drift will not change. In general, a task's drift per reweighting event will be non-negative (non-positive) if it increases (decreases) its weight. Under CNG-EDF, the drift of a task T_i at time t is defined as

$$\mathsf{drift}(T_i, t) = \mathsf{A}(\mathcal{I}, T_i, 0, u) - \mathsf{A}(\mathcal{S}W, T_i, 0, u),$$

where $\mathcal{S}W$ is the schedule of T under SW, $\mathcal{I}$ is the schedule of T under IDEAL, and u is the last time a reweighting event of T_i was enacted before t. (Notice that drift is defined in terms of SW instead of SW-NC. The reason for this is because SW-NC does not stop allocating capacity to a task after it has received its actual execution cost, and as a result SW-NC "over-allocates" capacity to tasks. SW on the other hand is always within a bounded range of a task's actual allocations.) Under CNG-EDF, the drift per reweighting event is bounded as follows.

Theorem 2. *The absolute value of the per-event drift under* CNG-EDF *for each task T_i is at most* $\mathsf{e}_{\mathsf{max}}(T_i)$.

Proof. We first show that for any job T_i^j of any task T_i $\mathsf{A}(\mathcal{S}W, T_i, \mathsf{r}(T_i^j), t_A) \leq \mathsf{e}_{\mathsf{max}}(T_i)$, where t_A is the time that T_i^j becomes inactive. Notice that since a job is inactive by its deadline (*i.e.*, $t_A \leq \mathsf{d}(T_i^j)$), and the deadline of T_i^j is defined as $\mathsf{r}(T_i^j) + \mathsf{e}(T_i^j)/\mathsf{swt}(T_i, \mathsf{r}(T_i^j))$, it follows that if T_i does not enact a weight change over the range $[\mathsf{r}(T_i^j), t_A)$, *i.e.*, T_i's scheduling weight is static over the range $[\mathsf{r}(T_i^j), t_A)$, then $\mathsf{A}(\mathcal{S}W, T_i, \mathsf{r}(T_i^j), t_A) \leq \mathsf{e}(T_i^j) \leq \mathsf{e}_{\mathsf{max}}(T_i)$.

Thus, in order for, $\mathsf{A}(\mathcal{S}W, T_i, \mathsf{r}(T_i^j), t_A) > \mathsf{e}_{\mathsf{max}}(T_i)$ to hold, it must be that T_i enacted a weight change over the range $[\mathsf{r}(T_i^j), t_A)$. Thus, we assume that T_i enacts a change over the range $[\mathsf{r}(T_i^j), t_A)$, and that t_e is the last such time. Notice that if the change enacted at t_e is by Rule P (i) or (ii) or Rule N (ii), then T_i^j becomes inactive at t_e, which contradicts our assumption that $t_e < t_A$. Thus, the change enacted at t_e must be by Rule N (i). By Rule N (i), T_i^j becomes inactive at the first time at or after t_e such that the deviance of T_i^j equals zero, *i.e.*, t_A is the smallest time such that $\int_{\mathsf{r}(T_i^j)}^{t_A} \mathsf{swt}(T_i, u)du = \mathsf{ae}(T_i^j)$. Since $\mathsf{A}(\mathcal{S}W, T_i, \mathsf{r}(T_i^j), t_A) \leq \int_{\mathsf{r}(T_i^j)}^{t_A} \mathsf{swt}(T_i, u)du$ and since $\mathsf{ae}(T_i^j) \leq \mathsf{e}(T_i^j) \leq \mathsf{e}_{\mathsf{max}}(T_i^j)$, it follows that

$$\mathsf{A}(\mathcal{S}W, T_i, \mathsf{r}(T_i^j), t_A) \leq \mathsf{e}_{\mathsf{max}}(T_i). \tag{3}$$

Let t_c be a time such that some task T_i initiates a weight change. Let t_e denote the next time that the change initiated at t_c is either canceled or enacted, whichever is first. Thus, by the definition of canceled, no weight change is initiated over the range (t_c, t_e). We show that regardless of which rule T_i uses to change its weight, the drift incurred by this initiation is at most $\mathsf{e}_{\mathsf{max}}(T_i)$. Let T_i^j be the last-released job of T_i before t_c. Notice that if T_i^j is not active at t_c or T_i^j does not exist, then the change is immediately enacted and no job is halted. Since the only two potential sources of drift are delays in enacting a weight change and halting a job (which causes the actual execution cost to be lower than the execution cost), it follows that if T_i^j does not exist or T_i^j is inactive at t_c then no drift is incurred. Thus, for the rest of this proof, we assume that T_i^j is active at t_c, and we let t_A denote the time that T_i^j becomes inactive. Notice by property (X), we have that

$$t_c \leq t_e \leq t_A \leq \mathsf{d}(T_i^j). \tag{4}$$

If T_i changes its weight at time t_c via Rule P (i), then since this weight change is immediately enacted (*i.e.*, $t_c = t_e$), it is as though allocation equal to $\mathsf{A}(\mathcal{I}, T_i, \mathsf{r}(T_i^j), t_c) - \mathsf{A}(\mathcal{SW}, T_i, \mathsf{r}(T_i^j), t_c)$ is "lost." For example in Fig. 7(a), the task T_4 "loses" an allocation of 2/6. Notice that per reweighting event $\mathsf{A}(\mathcal{I}, T_i, \mathsf{r}(T_i^j), t_c) - \mathsf{A}(\mathcal{SW}, T_i, \mathsf{r}(T_i^j), t_c) \leq \mathsf{e}_{\mathsf{max}}(T_i)$. Also note that since, by (3) and (4), $\mathsf{A}(\mathcal{SW}, T_i, \mathsf{r}(T_i^j), t_A) \leq \mathsf{e}_{\mathsf{max}}(T_i)$, it follows that $-\mathsf{e}_{\mathsf{max}}(T_i) \leq \mathsf{A}(\mathcal{I}, T_i, \mathsf{r}(T_i^j), t_c) - \mathsf{A}(\mathcal{SW}, T_i, \mathsf{r}(T_i^j), t_c)$. Thus, since $-\mathsf{e}_{\mathsf{max}}(T_i) \leq \mathsf{A}(\mathcal{I}, T_i, \mathsf{r}(T_i^j), t_c) - \mathsf{A}(\mathcal{SW}, T_i, \mathsf{r}(T_i^j), t_c) \leq \mathsf{e}_{\mathsf{max}}(T_i)$, it follows that the absolute value of drift is at most $\mathsf{e}_{\mathsf{max}}(T_i)$.

Suppose that T_i initiates a change to weight v via Rule P (ii) at v. Since this change is enacted or canceled at time t_e, it is as though allocation equal to $\mathsf{A}(\mathcal{I}, T_i, t_c, t_e) - \mathsf{A}(\mathcal{SW}, T_i, t_c, t_e)$ is "lost." Recall that a task only changes its weight via Case (ii) of Rule P if the deviance of T_i is positive and if $\mathsf{d}(T_i^j) - t_c \leq \mathsf{rem}(T_i^j, t_c)/v$. Since by (4), $t_e \leq \mathsf{d}(T_i^j)$ and $\mathsf{rem}(T_i^j, t_c) = \mathsf{e}_{\mathsf{max}}(T_i)$, the previous inequality can be rewritten as

$$v \cdot (t_e - t_c) \leq \mathsf{e}_{\mathsf{max}}(T_i). \tag{5}$$

Recall that no change is initiated over the range (t_c, t_e). Thus, by definition, $\mathsf{A}(\mathcal{I}, T_i, t_c, t_e) = \int_{t_c}^{t_e} v du = v \cdot (t_e - t_c)$. Hence, by (5), $\mathsf{A}(\mathcal{I}, T_i, t_c, t_e) \leq \mathsf{e}_{\mathsf{max}}(T_i)$. Furthermore, by (3) and (4), $\mathsf{A}(\mathcal{SW}, T_i, t_c, t_e) \leq \mathsf{e}_{\mathsf{max}}(T_i)$. Thus, since $-\mathsf{e}_{\mathsf{max}}(T_i) \leq \mathsf{A}(\mathcal{I}, T_i, t_c, t_e) - \mathsf{A}(\mathcal{SW}, T_i, t_c, t_e) \leq \mathsf{e}_{\mathsf{max}}(T_i)$, and $\mathsf{A}(\mathcal{I}, T_i, t_c, t_e) - \mathsf{A}(\mathcal{SW}, T_i, t_c, t_e)$ is lost per-reweighting event, it follows that the absolute value of drift is less than $\mathsf{e}_{\mathsf{max}}(T_i)$. For example, in Fig. 7(b), over the range $[3, 4)$ for task T_3, an allocation of $\mathsf{A}(\mathcal{I}, T_3, 3, 4) - \mathsf{A}(\mathcal{SW}, T_3, 3, 4) = 1/3 - 1/4 = 1/12$ is lost.

Suppose that T_i changes its weight to v at time t_c via Rule N. If T_i decreases its weight, then it is as though the allocation equal to $\mathsf{A}(\mathcal{I}, T_i, t_c, t_e) - \mathsf{A}(\mathcal{SW}, T_i, t_c, t_e)$ is lost. Furthermore, since it was a weight decrease initiated at t_c, it follows that

$\mathsf{A}(\mathcal{I}, T_i, t_c, t_e) < \mathsf{A}(\mathcal{S}W, T_i, t_c, t_e)$. Thus, since by (3) and (4), $\mathsf{A}(\mathcal{S}W, T_i, t_c, t_e) \le \mathsf{e}_{\mathsf{max}}(T_i)$, it follows that $\mathsf{A}(\mathcal{I}, T_i, t_c, t_e) < \mathsf{A}(\mathcal{S}W, T_i, t_c, t_e) \le \mathsf{e}_{\mathsf{max}}(T_i)$. Thus, since $-\mathsf{e}_{\mathsf{max}}(T_i) \le \mathsf{A}(\mathcal{I}, T_i, t_c, t_e) - \mathsf{A}(\mathcal{S}W, T_i, t_c, t_e) \le \mathsf{e}_{\mathsf{max}}(T_i)$ and $\mathsf{A}(\mathcal{I}, T_i, t_c, t_e) - \mathsf{A}(\mathcal{S}W, T_i, t_c, t_e)$ is lost per-reweighting event, it follows that the absolute value of the drift incurred is at most $\mathsf{e}_{\mathsf{max}}(T_i)$. For example, in Fig. 7(d), the drift incurred by T_4 is $-3/12$, *i.e.*, $\mathsf{drift}(T_4, t) = -3/12$, where $t \ge 3/2$. If T_i increases its weight (Case (i)), then it incurs zero drift, since it *immediately* enacts the weight change (*i.e.*, the scheduling weight changes immediately). Hence, the absolute value of the drift incurred by this reweighting event is less than $\mathsf{e}_{\mathsf{max}}(T_i)$. For example, in Fig. 7(c), the drift incurred by T_4 is 0, *i.e.*, $\mathsf{drift}(T_4, t) = 0$, where $t \ge 2$. $\square$

Notice that the presence of jobs that miss their deadlines does not affect the drift bounds. The reason for this is that the reweighting rules are *only* based on the state of the active job at the time the reweighting event is initiated. Thus, if a job has not been scheduled by the time it reweights, then it does not matter whether a predecessor prevented the job from being scheduled or the job had the lowest scheduling priority, only that the job has not been scheduled, and therefore is positive-changeable. Consider the example in Fig. 12. This example illustrates part of the actual, $\mathcal{S}W$, and $\mathcal{I}$ schedule for the task T_i that has the following properties: an initial weight of $1/10$ that increases to $1/2$ at time t_c; a job T_i^{j-1} that has a deadline at $t_r = t_c - 5$, an execution cost of 14, and misses its deadline by 11 quanta; and all jobs released after T_i^{j-1} have an execution cost of 1. Since T_i^{j-1} misses its deadline by 11 quanta and T_i^j is active for at most 10 quanta, T_i^j cannot execute while it is active in $\mathcal{S}$. However, when T_i changes its weight at time t_c (five quanta after T_i^j is released), T_i is positive-changeable, since T_i^j's deviance is positive. Thus, T_i changes its weight via Rule P with $1/2$ of a quantum of drift. Notice that this is the same amount of drift that T_i would have incurred if T_i^j was not scheduled before t_c because it had the lowest-scheduling priority.

Modifications for NP-CNG-EDF. Note that delaying the initiation of a reweighting event due to non-preemptivity does not substantially increase the drift incurred per reweighting event, since the longest a reweighting event can be delayed is the execution cost of the active job of the task being reweighted.

Suppose that the task T_i initiates a weight change at time t_c. If T_i^j is active at t_c, and if T_i's reweighting event is delayed until some time t (by a non-preemptive section), then at t either **(i)** T_i^j has a non-positive deviance (*i.e.*, T_i^j completes before its deadline), or **(ii)** t is the first time that T_i^j becomes inactive (*i.e.*, T_i^j misses its deadline and becomes inactive at t).

If Case (i) occurs, then T_i is negative-changeable at t, and T_i^j is active at t. Hence, if a task increases its weight, then the only drift the task will incur for this reweighting event results from delaying the initiation of the event, *i.e.*, at most $\mathsf{e}_{\mathsf{max}}(T_i)$. For example, consider Fig. 13(a), which depicts a task T_i of weight $1/10$ that increases its weight to $1/2$ at time t_c, where the

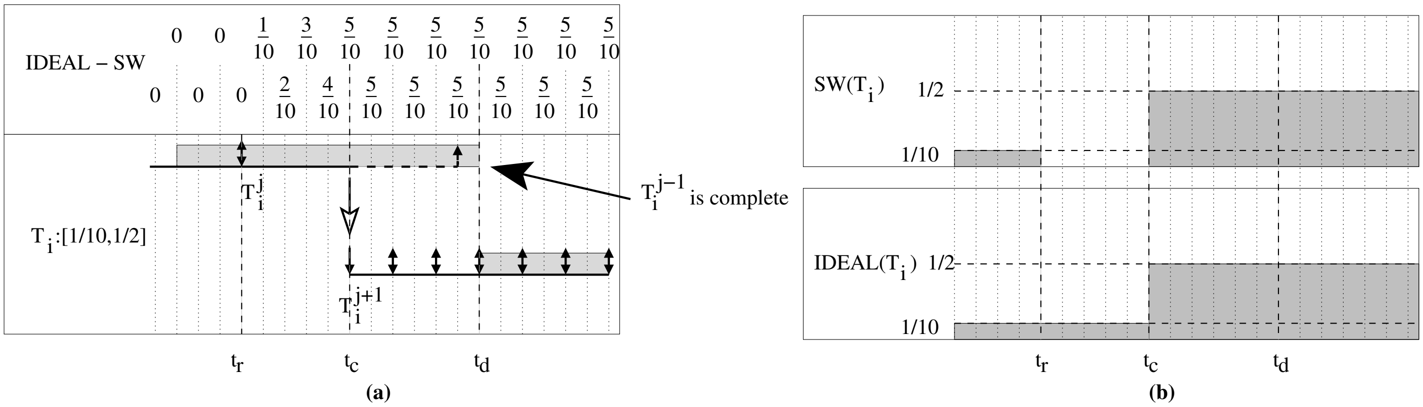


Figure 12: The partial **(a)** CNG-EDF, **(b)** SW and IDEAL schedules for the task T_i that has all of the following characteristics: an initial weight of $1/10$ that increases to $1/2$ at time t_c; a job T_i^{j-1} that has a deadline at $t_r = t_c - 5$, an execution cost of 14, and misses its deadline by 11 quantum; and all jobs released after T_i^{j-1} have an execution cost of 1. Notice that even though T_i^{j-1} misses its deadline at time t_r, when T_i initiates the change at time t_c, T_i^j is the active job, and since it has not been scheduled T_i is positive-changeable at t_c. Therefore, by Rule P, T_i^j (but *not* T_i^{j-1}) is halted and T_i^{j+1} is immediately released with the new weight, which incurs a drift of $1/2$.

last-released job of T_i before t_c, T_i^j, is active at t_c, being scheduled at t_c, and completes before $\mathsf{d}(T_i^j)$. In this example, the only drift that T_i incurs when its weight increases from $1/10$ to $1/2$ is because the weight change initiation must be delayed for three time units. If T_i decreases its weight, then delaying the reweighting event will not affect drift, since the enactment of the reweighting event would occur when T_i^j becomes inactive, regardless of whether the initiation of the reweighting event was delayed or not.

If Case (ii) occurs, then either no job of T_i is active at t or T_i^{j+1} is active at t. If no job of T_i is active at t, then the change is enacted immediately, and the drift that the task incurs from the reweighting event is a result from delaying the initiation of the event, *i.e.*, $\mathsf{e}_{\mathsf{max}}(T_i)$. If T_i^{j+1} is active at t, then since t is the first time that T_i^j is inactive after it was released, it must be the case that $\mathsf{r}(T_i^{j+1}) = t$. As a result, the weight change is enacted immediately and T_i^{j+1} is released with the new weight. Hence, the only drift that is incurred is as a result of delaying the initiation of the reweighting event, *i.e.*, at most $\mathsf{e}_{\mathsf{max}}(T_i)$. For example, consider the system in Fig. 13(b), which depicts a task T_i of weight $1/10$ that increases its weight to $1/2$ at time t_c, where the last-released job of T_i before t_c, T_i^j, is active at t_c, is being scheduled at t_c, and completes *after* $\mathsf{d}(T_i^j)$. Thus, the weight increase is delayed such that it is initiated and enacted at $\mathsf{d}(T_i^j)$, and the only drift that is incurred is because the initiation of the reweighting event is delayed for two quantum. From this logic we can see that the following theorem holds.

Theorem 3. *The absolute value of the per-event drift under* NP-CNG-EDF *for each task T_i is at most* $\mathsf{e}_{\mathsf{max}}(T_i)$.

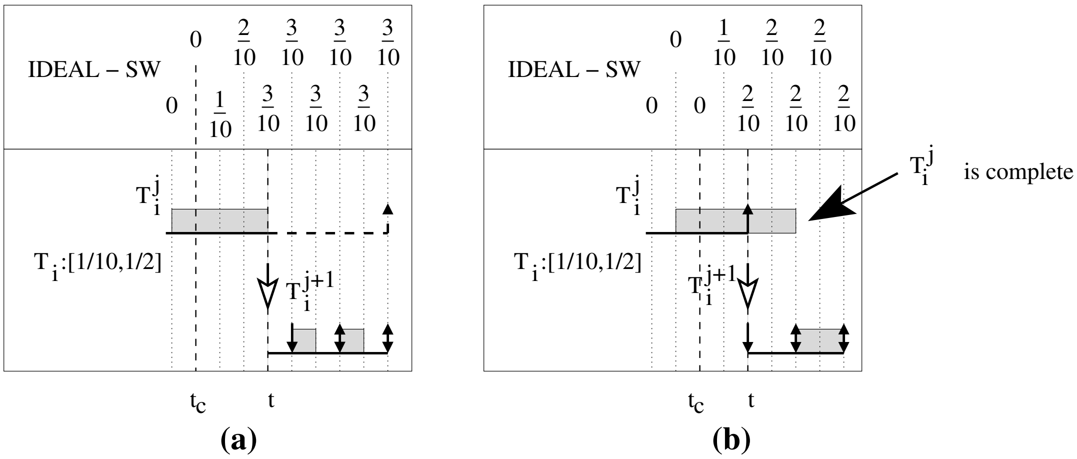


Figure 13: A partial NP-CNG-EDF schedule that contains the task T_i, which has an initial weight of $1/10$ that increases to $1/2$ at time t_c while the last-released job of T_i before t_c, T_i^j, is both active and being scheduled. Note that T_i^j has an execution cost of 4, and all jobs released after T_i^j have an execution cost of 1. **(a)** The scenario where T_i^j completes execution five quantum before its deadline at time t. As a result, the initiation of this reweighting event is delayed from t_c until time t. Since T_i^j is the last-released job of $T_i j$ before t_c, it is active at t, and its deviance is negative, T_i is negative-changeable. Thus, by Rule (N), T_i^{j+1} is released when its deviance equals zero, *i.e.*, $t+1$. Notice that the only source of drift from this reweighting event is delaying the initiation of the reweighting event. **(b)** The scenario where T_i^j does not complete execution until after its deadline. As a result, the initiation of the weight change is delayed until $t = \mathsf{d}(T_i^j) = \mathsf{r}(T_i^{j+1})$. Recall that if a weight change is initiated when $\mathsf{d}(T_i^j) = \mathsf{r}(T_i^{j+1})$, then the weight change is immediately enacted and T_i^{j+1} is released with the new weight. Thus, the only source of drift is because the initiation of the reweighting event is delayed.

5 Experimental Results

The results of this paper are part of a longer-term project on adaptive real-time allocation in which both Whisper and ASTA described earlier, will be used as test applications. In this section, we provide extensive simulations of Whisper and ASTA as scheduled by PD^2-OI, PAS, NP-PAS, CNG-EDF, and NP-CNG-EDF.

Whisper. As noted earlier, Whisper tracks users via speakers that emit white noise attached to each user's hands, feet, and head, as depicted in Fig. 14. Microphones located on the wall or ceiling receive these signals and a tracking computer calculates each speaker's position by measuring signal delays. 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

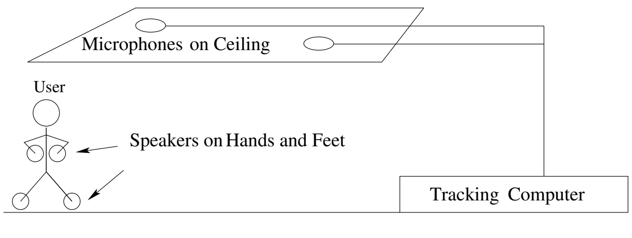


Figure 14: The Whisper system.

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 10ms. 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/preemption 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 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 the 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/preemption 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 more 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-OI, PAS, NP-PAS, CNG-EDF, and NP-CNG-EDF 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 virtual-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 100 times. Both systems were simulated for 10 secs. (Note that decreasing and increasing the simulation time gives similar results.) 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 5μs. We considered this value to be negligible in comparison to a 1-ms quantum and thus did not consider scheduling overheads in our simulations. For both Whisper and ASTA, we conducted two types of experiments: **(i)** all preemption and migration costs were the same and corresponded to a loss of cache affinity; and **(ii)** the preemption cost was set to some value and the migration cost was varied. If a task was preempted and then migrated, we assumed that it incurred the maximum of the two costs. 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 [9]. Staggering would be trivial to apply in PAS and NP-PAS as well, since in PAS, processors run nearly independently of each other. Furthermore, since CNG-EDF and NP-CNG-EDF are event-based rather than quantum-based, jobs are unlikely to begin executing simultaneously. Based on measurements taken on our testbed system, we estimated Whisper's migration cost as 2μs–10μs, and ASTA's as 50μs–60μs. While we believe that these costs may be typical for a wide range of systems, in our experiments we varied the preemption/migration cost over a slightly larger range. For all experiments, the maximum execution cost was 7ms for PAS and NP-PAS and 5ms for CNG-EDF and NP-CNG-EDF. These values were determined by profiling each system beforehand to determine the "best" compromise of accuracy and performance.

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 allocations in the IDEAL algorithm. In particular, we measured both the "average under-allocation" and "fairness factor" for each task set at the end of each simulation (*i.e.*, 10 secs.). The *average under-allocation* (UA) is 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 *fairness factor* (FF) of a task set is the largest deviance from the allocations in IDEAL between any two tasks (*e.g.*, if a system has three tasks, one that deviates from its IDEAL allocation 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 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 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.

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. 15. (The results of these simulations appear in Fig. 16 and Fig. 17.) 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.

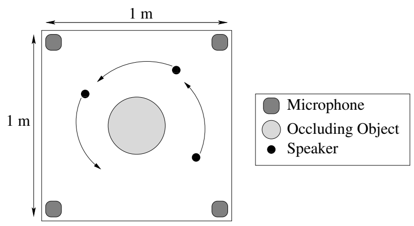


Figure 15: The simulated Whisper system.

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 5cm 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

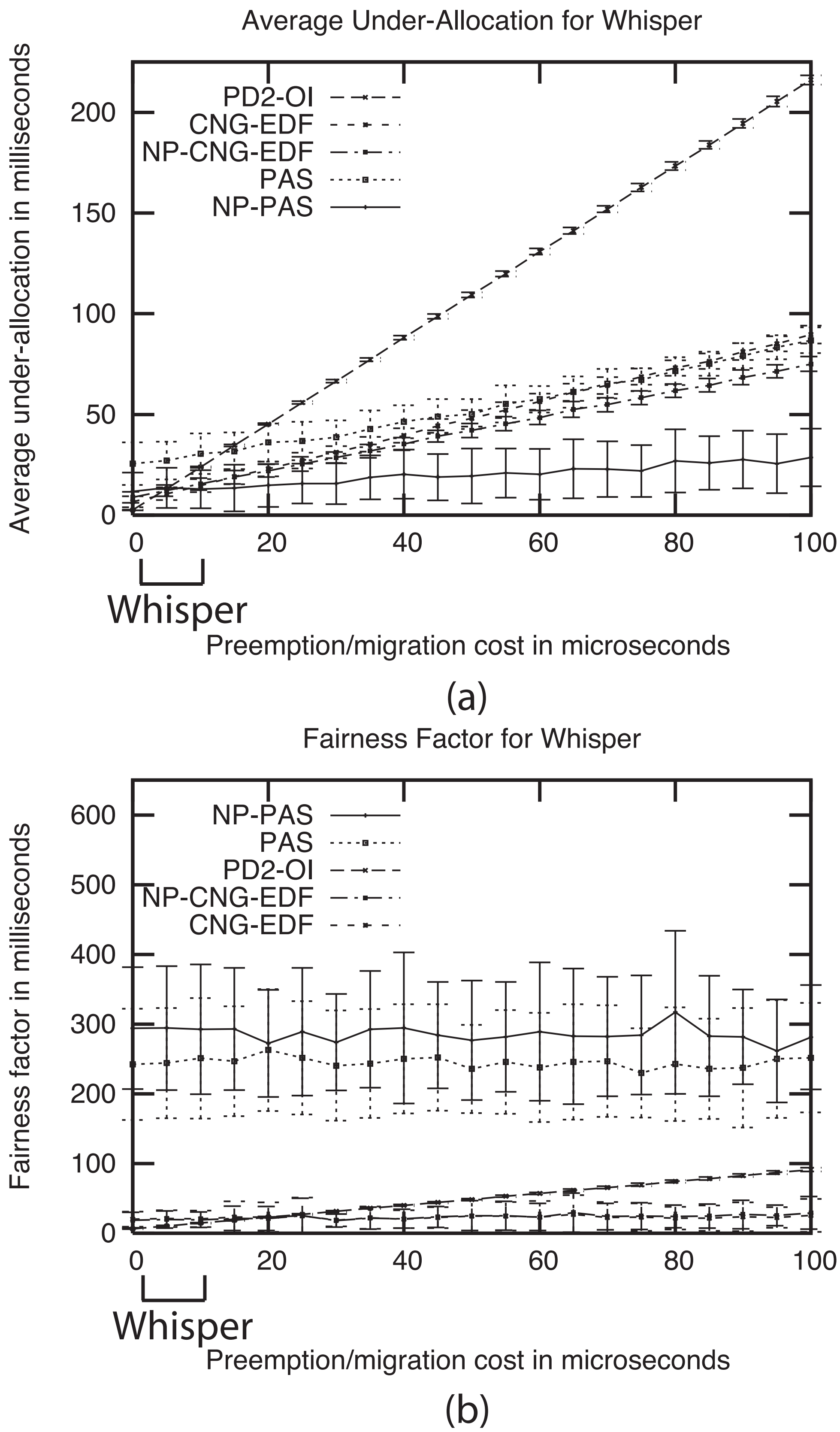
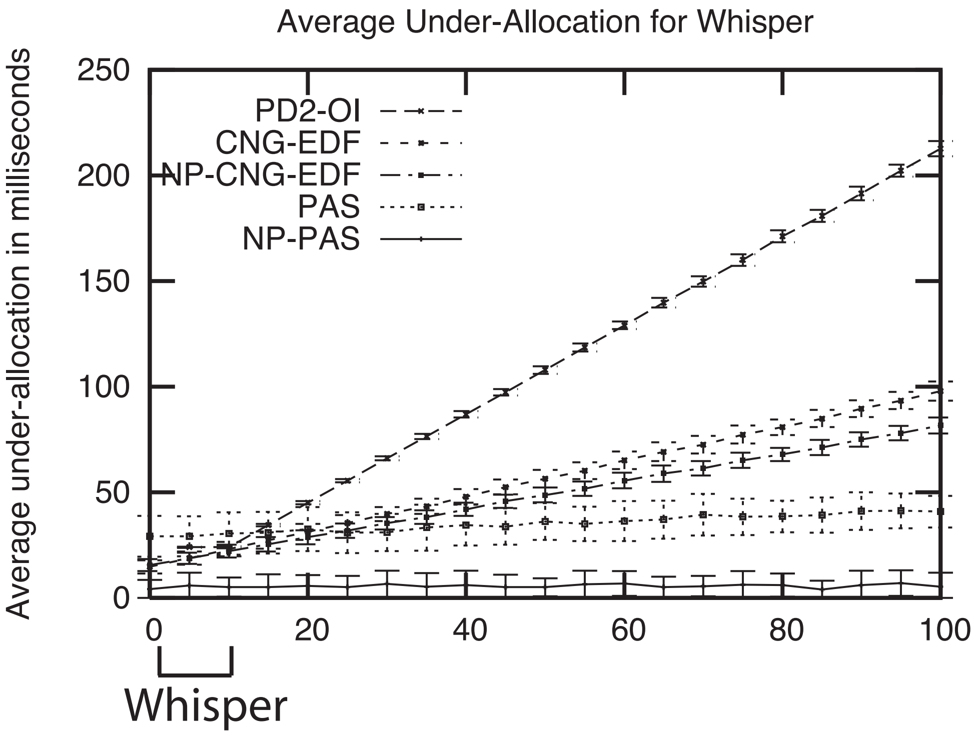
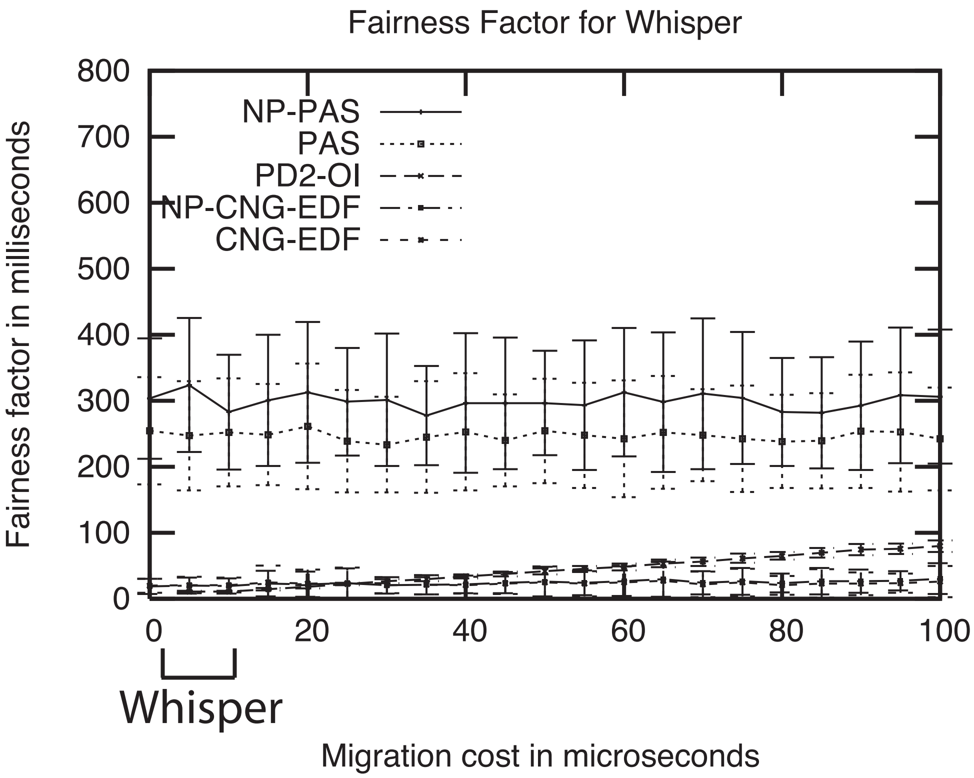


Figure 16: **(a)** The average under-allocation (UA) and **(b)** the fairness factor (FF) for Whisper as a function of preemption/migration cost for Whisper as a function of migration cost (preemption cost is fixed at 10μs), as scheduled by each tested algorithm. The key in each graph is in the order that the schemes appear in that graph at 100μs. Standard deviations are shown. Note that in (b) CNG-EDF and NP-CNG-EDF are indistinguishable from each other.



(a)



(b)

Figure 17: **(a)** The average UA and **(b)** FF for Whisper as a function of preemption/migration cost for Whisper as a function of migration cost (preemption cost is fixed at 10μs), as scheduled by each tested algorithm. The key in each graph is in the order that the schemes appear in that graph at 100μs. Standard deviations are shown. Note that in (b) CNG-EDF and NP-CNG-EDF are indistinguishable from each other.

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 50cm, the speed of each object was set at 5 m/sec. (this is within the speed of human motion), and the maximum execution cost, migration, and preemption cost were varied.

The first set of graphs in Fig. 16 and Fig. 17 show the result of the Whisper simulations conducted to compare PD^2-OI, PAS, NP-PAS, CNG-EDF, and NP-CNG-EDF. Fig. 16 depicts the average UA and FF, respectively, for each scheme, where the preemption cost is varied from 0 to 100μs and the migration cost equals the preemption cost. Fig. 17 depicts the average UA and FF, respectively, for each scheme, where the preemption cost is set at 10μs (the maximum expected preemption cost for Whisper) and the migration cost is varied from 0 to 100μs. There are five things worth noting here. First, when the preemption/migration cost is varied over the range 2 to 10μs, the UA is about the same for all schemes (Fig. 16(a)); however, PD^2-OI has the best FF (Fig. 16(b)). Second, while CNG-EDF and NP-CNG-EDF do not have the best UA for the expected preemption/migration costs for Whisper, for higher preemption/migration costs, *i.e.*, preemption/migration costs larger than 10μs, CNG-EDF and NP-CNG-EDF both have a substantially better UA than PD^2-OI and better FF than either PAS or NP-PAS. Third, as the migration cost (but not preemption cost) of a task increases, the UA of PAS and NP-PAS increases slowly (Fig. 17(a)). However the performance of the other three schemes decays quickly. Fourth, standard deviations of the FF for CNG-EDF, NP-CNG-EDF, and PD^2-OI are smaller than for PAS and NP-PAS, since CNG-EDF, NP-CNG-EDF, and PD^2-OI have better accuracy. Fifth, Fig. 17, PD^2-OI and CNG-EDF's UA and FF do not appreciably increase until the migration cost exceeds 10μs. This is because, until the migration cost is 10μs, PD^2-OI and CNG-EDF incur the maximum of the migration or preemption cost, which is 10μs.

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. 18. 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

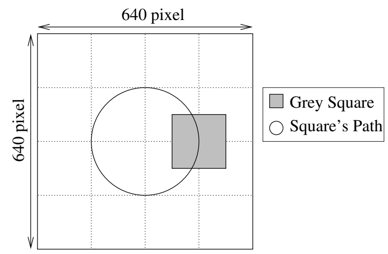


Figure 18: The simulated ASTA system.

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 40ms.

The second set of graphs, in Fig. 19 and Fig. 20, show the result of the ASTA simulations conducted to compare the five scheduling algorithms. Fig. 19 depicts the average UA and FF, for each scheme, where the preemption cost is varied from 0 to 100μs and the migration cost equals the preemption cost. Fig. 20 depicts the average UA and FF for each scheme, where the preemption cost is set at 60μs (the maximum expected preemption cost for ASTA) and the migration cost is varied from 0 to 100μs. There are two things worth noting here. First, when the preemption/migration cost is varied over the range 50 to 60μs, NP-PAS and PAS have the smallest UA (Fig. 19(a)); however, CNG-EDF and NP-CNG-EDF both have a UA that is competitive with both PAS and NP-PAS (Fig. 19(a)) and have a *substantially* smaller FF (Fig. 19(b)). Second, in Fig. 20, PD^2-OI and CNG-EDF's UA and FF do not appreciably increase until the migration cost equals 60μs. This occurs for the same reason that PD^2-OI and CNG-EDF did not noticeably increase, until 10μs in Fig. 17.

6 Concluding Remarks

We have presented two new multiprocessor reweighting schemes, CNG-EDF and NP-CNG-EDF, which reduce migration costs and preemptions at the expense of allowing deadline misses. We have also presented both analytical and experimental comparisons of these schemes with a more accurate but more migration-prone scheme, PD^2-OI, and two less accurate partitioning schemes that have lower tardiness, PAS and NP-PAS. These results suggest that when it is critical that every task make its deadline and migration/preemption costs are low (*i.e.*, systems like Whisper), PD^2-OI is the best choice; when preemption/migration costs are high (*i.e.*, either Whisper or ASTA as implemented on a system where the processors are not as tightly integrated), average case performance is of the utmost importance, and fairness and timeliness are less important, then either PAS or NP-PAS may be the best choice; and when preemption/migration costs are high and a good mix of average-case performance and fairness factor is beneficial (*i.e.*, systems like ASTA), then either CNG-EDF or NP-CNG-EDF may be the best choice. Thus, *each algorithm is of value* and will be the best choice in certain application scenarios, as summarized in Table 4.

While our focus in this paper has been on scheduling techniques that *facilitate* fine-grained adaptations of weight and execution cost, 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

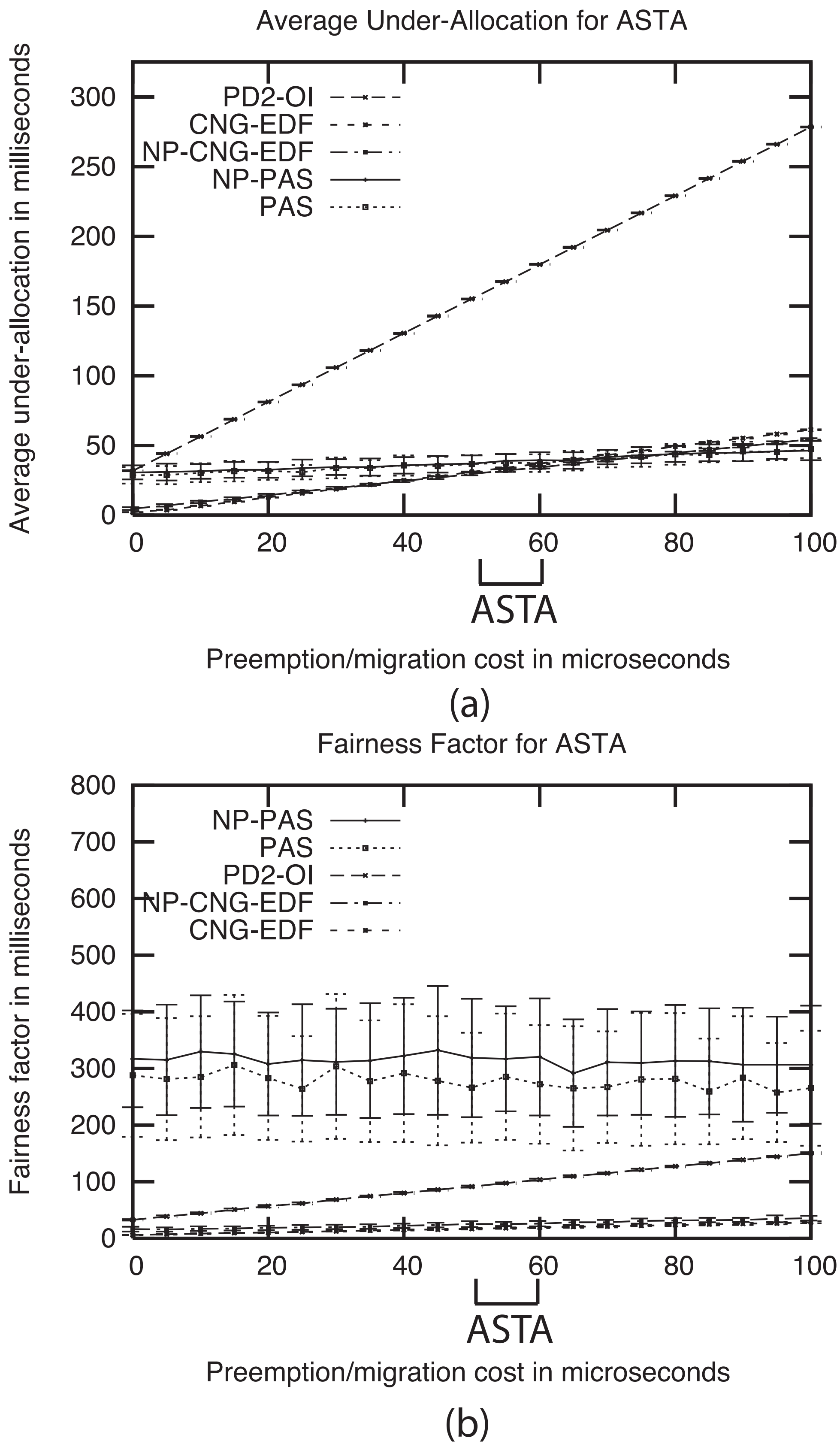
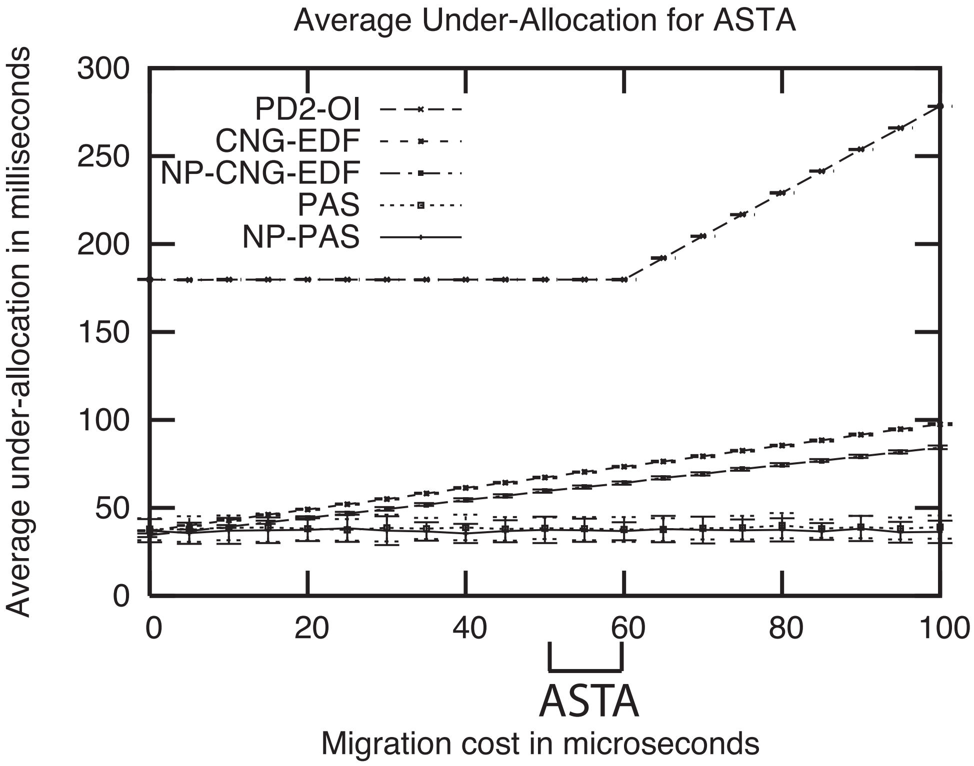
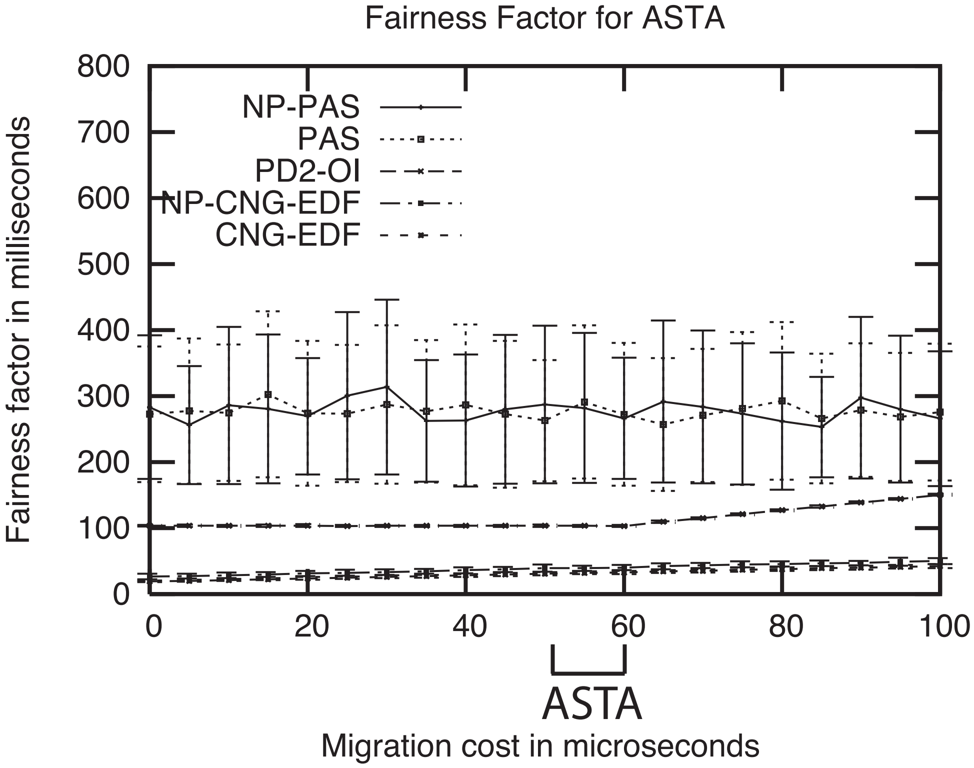


Figure 19: **(a)** The average under-allocation (UA) and **(b)** the fairness factor (FF) for ASTA as a function of preemption/migration cost as a function of migration cost (preemption cost is fixed at $60\mu s$), as scheduled by each tested algorithm. The key in each graph is in the order that the schemes appear in that graph at $100\mu s$. Standard deviations are shown. Note that in (b), CNG-EDF and NP-CNG-EDF are indistinguishable from each other.



(a)



(b)

Figure 20: **(a)** The average UA and **(b)** FF for ASTA as a function of migration cost (preemption cost is fixed at 60μs), as scheduled by each tested algorithm. The key in each graph is in the order that the schemes appear in that graph at 100μs. Standard deviations are shown. Note that in (b), CNG-EDF and NP-CNG-EDF are indistinguishable from each other.

Scheme	Provides Hard Real-Time Guarantees	Has Low Migration/ Preemption Costs	Provides Strong Fairness Guarantees
PD^2-OI	✓		✓
(NP-)PAS		✓	
(NP-)CNG-EDF		✓	✓

Table 4: Summary of algorithm performance.

incorporated within scheduling algorithms [11]). 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. *Journal of Embedded Computing*, special issue on multiprocessor real-time scheduling, to appear.

[4] 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.

[5] A. Block and J. Anderson. Accuracy versus Migration Overhead in Multiprocessor Reweighting Algorithms. In *Proceedings of the 12th International Conference on Parallel and Distributed Sys.*, July 2006, to appear.

[6] 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.

[7] U. Devi and J. Anderson. Tardiness Bounds for Global EDF Scheduling on a Multiprocessor. In *Proceedings of the 26th IEEE Real-Time System Symposium*, pages 330-341. December 2005.

[8] U. Devi and J. Anderson. Tardiness Bounds for Global EDF Scheduling on a Multiprocessor. In submission. Available at `http://www.cs.unc.edu/~anderson/papers/`.

[9] 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.

[10] 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.

[11] 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.

[12] A. Srinivasan and J. Anderson. Fair Scheduling of Dynamic Task Systems on Multiprocessors. In *Journal of Systems and Software*, Volume 77, Number 1, pp. 67-80, April 2005.

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

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