Using Supertasks to Improve Processor Utilization in Multiprocessor Real-time Systems*

Philip Holman and James H. Anderson

Department of Computer Science

University of North Carolina

Chapel Hill, NC 27599-3175

E-mail: {holman,anderson}@cs.unc.edu

December 2002

Abstract

We revisit the problem of supertasking in Pfair-scheduled multiprocessor systems. In this approach, a set of tasks, called *component tasks*, is assigned to a server task, called a *supertask*, which is then scheduled as an ordinary Pfair task. Whenever a supertask is scheduled, its processor time is allocated to its component tasks according to an internal scheduling algorithm. Hence, supertasking is an example of hierarchal scheduling.

In this paper, we present a generalized "reweighting" algorithm. The goal of reweighting is to assign a fraction of a processor to a given supertask so that all timing requirements of its component tasks are met. The generalized reweighting algorithm we present breaks new ground in three important ways. First, component tasks are permitted to have non-integer execution costs. Consequently, supertasking can now be used to ameliorate schedulability loss due to the *integer-cost* assumption of Pfair scheduling. To the best of our knowledge, no other techniques have been proposed to address this problem. Second, blocking terms are included in the analysis. Since blocking terms are used to account for a wide range of behaviors commonly found in actual systems (*e.g.*, non-preemptivity, resource sharing, deferrable execution, *etc.*), their inclusion is a necessary step towards utilizing supertasks in practice. Finally, the scope of supertasking has been extended by employing a more flexible global-scheduling model. This added flexibility allows supertasks to be utilized in systems that do not satisfy strict Pfairness. To demonstrate the efficacy of the supertasking approach, we present an experimental evaluation of our algorithm that suggests that reweighting may often result in *almost no* schedulability loss in practice.

*Work supported by NSF grants CCR 9972211, CCR 9988327, and ITR 0082866.

1 Introduction

Multiprocessor scheduling techniques fall into two general categories: *partitioning* and *global scheduling*. In the partitioning approach, each processor schedules tasks independently from a local ready queue. In contrast, all ready tasks are stored in a single queue under global scheduling and interprocessor migration is allowed. Presently, partitioning is the favored approach, largely because well-understood uniprocessor scheduling algorithms can be used for per-processor scheduling. Despite its popularity, the partitioning approach is inherently suboptimal when scheduling periodic tasks. A well-known example of this is a two-processor system that contains three synchronous periodic tasks, each with an execution cost of 2 and a period of 3. Completing each job before the release of its successor is impossible in such a system without migration.

One particularly promising global-scheduling approach is *proportionate-fair* (Pfair) scheduling, first proposed by Baruah *et al.* [1]. Pfair scheduling is presently the only known optimal method for scheduling recurrent tasks in a multiprocessor system. Under Pfair scheduling, each task is assigned a *weight*, w, that specifies the rate at which it should execute. Tasks are scheduled using a fixed-size allocation quantum so that each task's actual allocation over any time interval $[0, L)$ is strictly within one quantum of its ideal allocation, $w \cdot L$.

Unfortunately, in prior work on Pfair scheduling, it is assumed that all execution costs are expressed as multiples of the quantum size. Consequently, the theoretical optimality of Pfair scheduling hinges on the assumption that the quantum size can be made arbitrarily small. In actual systems, practical factors, such as context-switching and scheduling overhead, impose an implicit lower bound on the quantum size. When the quantum size is restricted, Pfair scheduling algorithms are *not* optimal. For instance, a periodic task with a worst-case execution time of 11 *ms* would need to reserve 20 *ms* of processor time in a Pfair-scheduled system that uses a 10-*ms* quantum. Such an inflated reservation results in considerable schedulability loss. Although context-switching and migration overhead is the most common criticism of Pfair scheduling, recent experimental work [7] suggests that this *integer-cost* problem is actually a more serious concern.

Hybrid Approach. In recent work, we investigated the use of *supertasks* in Pfair-scheduled systems [3]. Under supertasking, a set of tasks, called *component tasks*, is assigned to a server task, called a *supertask*, which is then scheduled as an ordinary Pfair task. Whenever a supertask is scheduled, its processor time is allocated to its component tasks according to an internal scheduling algorithm.

Although a supertask should ideally be assigned a weight equal to the cumulative utilization of its component tasks, Moir and Ramamurthy [5] demonstrated that the timing requirements of component tasks may be violated when using such an assignment. In prior work [3], we proved that such violations can be avoided by using a more pessimistic weight assignment. Unfortunately, this prior work suggests that reweighting will almost always result in schedulability loss. However, empirical data presented later suggests that this loss should be very small (and often *negligible*) in practice.

Although conceptually similar to partitioning, supertasking provides more flexibility when selecting task assignments. Whereas the number of processors and their capacities are fixed under partitioning, the number of supertasks and their capacities are not. Indeed, supertask capacities (*i.e.*, weights) can be assigned *after* making task assignments. This weight-assignment problem is referred to as the *reweighting* problem. For the purpose of assigning tasks, each supertask can temporarily be granted a unit capacity (weight). Following the assignment of tasks, these weights can then be reduced to minimize schedulability loss.

Furthermore, supertasking can be used to ameliorate schedulability loss due to the integer-cost problem. By packing tasks into supertasks and using a uniprocessor scheduling algorithm that does *not* require integer execution costs, loss due to the integer-cost problem can be traded for loss due to reweighting. Later, we present empirical evidence that shows the efficacy of this trade-off. Indeed, the results presented later suggest that reweighting may often result in *almost no* schedulability loss in practice.

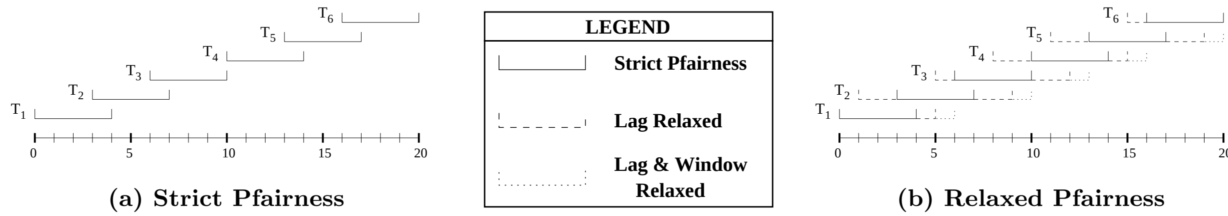
Contributions of this Paper. Unfortunately, the theory of supertasking has not been sufficiently developed to permit its use in practice. In prior work on supertasking [3, 5], it has been assumed that the global scheduler respects strict Pfairness, that component tasks have integer execution costs, and that component tasks are independent. These latter two assumptions are obviously unrealistic. Furthermore, there has been recent interest in applying Pfair scheduling with relaxed scheduling constraints [2, 8]. This interest stems from the fact that relaxing scheduling constraints typically permits the use of simpler algorithms, which, in turn, reduces overhead. For instance, our prior work on supertasks proved that allowing component tasks to experience bounded deadline misses can result in substantial schedulability gains [3].

In this paper, we present a generalized reweighting algorithm that extends our prior work on reweighting in the following three ways: **(i)** execution costs are no longer assumed to be integral; **(ii)** the global-scheduling model supports the use of relaxed fairness constraints; **(iii)** blocking terms are included in the analysis. In addition to the reasons given above, these three extensions are important because many task behaviors commonly encountered in practice, such as blocking and self-suspension, can disrupt fair allocation rates and hence are problematic under fair scheduling. By utilizing supertasks, tasks exhibiting such behaviors can be grouped together and scheduled using *unfair* uniprocessor algorithms, resulting in considerably less schedulability loss.

The remainder of the paper is organized as follows. Sec. 2 summarizes background information and notational conventions. In Sec. 3, the reweighting condition used in our analysis is derived. Our reweighting algorithm is then presented and analyzed in Secs. 4 and 5. Sec. 6 contains experimental results that demonstrate the efficacy of using supertasks to reduce schedulability loss. We conclude in Sec. 7.

2 Background

Given a set of component tasks τ, our goal is to compute the smallest weight that can be assigned to the corresponding supertask $\mathcal{S}$ so that the schedulability of τ is ensured. (We will precisely define the schedulability



(a) Strict Pfairness **(b) Relaxed Pfairness**

Figure 1: The first six windows of a task with weight $T.w = 3/10$ are shown up to time 20 for **(a)** strict Pfairness and **(b)** relaxed Pfairness where $\ell = 1.5$, $\alpha_r = 0$, and $\alpha_d = 1$.

requirement below.) This reweighting process requires consideration of both the global scheduler and the supertask's internal scheduler. We consider each in turn.

Global Scheduling. We begin by explaining the Pfairness constraint, and then discuss how it can be relaxed. Under Pfair scheduling, each task T is given a *weight* $T.w$ in the range (0,1]. Conceptually, $T.w$ is the rate at which T should be executed. Processor time is allocated in discrete time units, called *quanta*. The interval $[t, t+1)$, where t is a non-negative integer, is called *time slot* t. In each time slot, each processor can be assigned to at most one task and each task can be assigned at most one processor. Task migration is allowed.

The Pfairness constraint, first proposed by Baruah *et al.* [1], is defined by comparing the amount of processor time received by each task to the amount that would have been received in an ideal system. This concept is formalized by the notion of *lag*. Since a task T with weight $T.w$ would ideally receive $T.w \cdot L$ units of processor time in the time interval $[0, L)$, the *lag* of T at t can be formally defined as $\mathit{lag}(T, t) = T.w \cdot t - \mathit{allocated}(T, 0, t)$, where $\mathit{allocated}(T, a, b)$ denotes the amount of processor time allocated to task T over the time interval $[a, b)$. The Pfairness constraint can then be stated as follows:

$$(\forall T, t :: |\mathit{lag}(T, t)| < 1). \tag{1}$$

Informally, each task's allocation error must always be within one quantum. In [1], Baruah *et al.* proved that a schedule satisfying Condition (1) exists iff the sum of all task weights is at most the number of processors.

Condition (1) effectively sub-divides each task T into a series of quantum-length *subtasks*. Let T_i denote the ith subtask of T. Fig. 1(a) shows the time interval within which each subtask must execute to satisfy (1) when $T.w = 3/10$. The interval associated with each subtask T_i is called T_i's *window*. For example, the window of T_2 in Fig. 1(a) is $[3, 7)$. T_i's window is said to extend from its *release*, denoted $r(T_i)$, to its *deadline*, denoted $d(T_i)$. For example, $r(T_2) = 3$ and $d(T_2) = 7$ in Fig. 1(a). Hence, a schedule respects Pfairness iff each subtask T_i is scheduled in the time interval $[r(T_i), d(T_i))$.

Now, consider the problem of expressing a relaxed constraint. (Recall that weaker constraints can be used to reduce runtime overhead.) There are two obvious ways to do this. The first is to relax the lag bound in (1) by replacing 1 with a system-specific parameter $\ell \geq 1$. Such a change effectively scales each subtask window. However, the use of a scheduling quantum causes the windows to be truncated so that they begin and end only on slot boundaries. As a result, this scaling may not be uniform across all subtasks, as shown in Fig. 1(b). In this figure, $\ell = 1.5$. Notice that the windows of T_2, T_3, and T_4 are each affected differently by the scaling.

Alternatively, the Pfairness constraint can be relaxed by applying a uniform extension to all windows. Let α_r and α_d denote the maximum number of slots by which each window's release and deadline, respectively, are extended. (Assume that α_r and α_d are non-negative integers.) Fig. 1(b) illustrates the case in which $\ell = 1.5$, $\alpha_r = 0$, and $\alpha_d = 1$. Note that each window's deadline is extended one slot beyond its lag-based placement. To avoid confusion, we will refer to the window obtained by using α_r and α_d as the *extended window* of the subtask. For example, T_2 has a deadline at time 9 and an extended deadline at time 10 in Fig. 1(b). As shown below, only the sum of these parameters, $\alpha = \alpha_r + \alpha_d$, will be important when reweighting.

The following theorem formally expresses the impact ℓ and α have on the amount of time available to component tasks over any interval of time.

Theorem 1 *Any task T with weight $T.w$ executing in a relaxed Pfair-scheduled system characterized by ℓ, α_r, and α_d will receive at least* $\max(0, \lfloor T.w \cdot (L - \alpha) - 2\ell \rfloor + 1)$ *quanta of execution time over each slot interval of length L, where L is any non-negative integer and $\alpha = \alpha_r + \alpha_d$.*

Proof. When $\alpha_r = \alpha_d = 0$, the lag bound implies that $|allocated(T, 0, u) - T.w \cdot u| < \ell$ must hold at every time u. Since $allocated(T, 0, u)$ is always an integer at slot boundaries, this bound can be rewritten as shown below when u corresponds to a slot boundary.

$$\lfloor T.w \cdot u - \ell \rfloor + 1 \leq allocated(T, 0, u) \leq \lceil T.w \cdot u + \ell \rceil - 1$$

Informally, $\lfloor T.w \cdot u - \ell \rfloor + 1$ (respectively, $\lceil T.w \cdot u + \ell \rceil - 1$) is the number of subtasks with deadlines at or before (respectively, with releases before) *integer* time u.

However, a subtask with a deadline at d that is scheduled under relaxed window constraints may not complete until its extended deadline at time $d + \alpha_d$. Thus, subtasks with deadlines after $u - \alpha_d$ should not be counted in the lower bound. Similarly, subtasks with releases before $u + \alpha_r$ may have been scheduled before u and hence should be counted in the upper bound. These adjustments yield the following bounds:

$$\lfloor T.w \cdot (u - \alpha_d) - \ell \rfloor + 1 \leq allocated(T, 0, u) \leq \lceil T.w \cdot (u + \alpha_r) + \ell \rceil - 1. \qquad (2)$$

The derivation given below completes the proof.

$$\begin{aligned} & allocated(T, t, t + L) \\ & \quad = allocated(T, 0, t + L) - allocated(T, 0, t) && \text{, by definition} \\ & \quad \geq [\lfloor T.w \cdot (t + L - \alpha_d) - \ell \rfloor + 1] - [\lceil T.w \cdot (t + \alpha_r) + \ell \rceil - 1] && \text{, by (2)} \end{aligned}$$

$$= \lfloor T.w \cdot (t + \alpha_r) + \ell + T.w \cdot (L - \alpha) - 2\ell \rfloor - \lceil T.w \cdot (t + \alpha_r) + \ell \rceil + 2 \quad \text{, rewriting}$$
$$\geq \lfloor T.w \cdot (L - \alpha) - 2\ell \rfloor + 1 \quad , \lfloor a + b \rfloor \geq \lceil a \rceil + \lfloor b \rfloor - 1 \quad \square$$

Local Scheduling. We assume that τ (the set of component tasks under consideration) consists solely of N *periodic* or *sporadic* tasks. Each sporadic (respectively, periodic) task T is characterized by the tuple (θ, e, p), where $T.e$ is the execution requirement of each job, $T.p$ is the minimum (respectively, exact) separation between consecutive jobs releases, and θ is the release time of the first job. (θ is mentioned here only for completeness; it will not be needed in our analysis.) We assume that all job releases and deadlines of component tasks coincide with slot boundaries. We further assume that each such task has a *tardiness threshold* $T.c$, which is the allowable difference between a job's completion time and its deadline. Hence, T's jobs have *hard* deadlines when $T.c = 0$ and *soft* deadlines otherwise. (Whereas relaxed fairness constraints can be exploited to reduce global-scheduling overhead, tardiness thresholds can be used to reduce the schedulability loss caused by reweighting.) Based on these parameters, we can formally state the component-task schedulability condition. τ is said to be *schedulable* by $\mathcal{S}$ with respect to some component-task scheduling algorithm σ (defined below) if and only if for all tasks T in τ and for all jobs of T, the tardiness of the job does not exceed $T.c$ in the schedule produced by σ.

We now define σ. A job of task T with an absolute deadline at d can be considered to have an *effective deadline* at time $d + T.c$ due to the use of tardiness thresholds. Hence, an earliest-effective-deadline-first (EEDF) policy is the obvious choice for σ. Under this policy, jobs with earlier effective deadlines are given higher priority.

To simplify the presentation of our results, we define a few shorthand notations. Let $T.u = T.e/T.p$ denote the *utilization* of T, and let U denote the sum of these values across all tasks in τ. Also, let $\tau(L)$ denote the subset of τ consisting only of tasks with tardiness thresholds at most L.

3 Reweighting Problem

In this section, we present the reweighting problem and discuss some of its characteristics.

Deriving a Reweighting Condition. Suppose that τ is scheduled using EEDF priorities and that some job J of task T both violates its tardiness threshold and is the first job to do so. Let t_r (respectively, t_d) be the release (respectively, deadline) of J. Due to the violation, there must exist an interval $I = [t, t + L)$, where $t \leq t_r$ and $t + L = t_d + T.c$, such that the total demand for processing time exceeded the available processing time, as expressed in (3).

$$demand(\tau, T, I) > available(I) \tag{3}$$

For simplicity, we henceforth assume that t falls on a slot boundary, which implies that L is an integer.[1]

[1] In most cases, this assumption should hold. When it does not, we can still derive a necessary condition for a timing violation of the form given by (3) by manipulating the form of the blocking term, which is introduced below.

Under EEDF scheduling, $demand(\tau, T, I)$ can be expressed as a combination of two terms. First, processor time is demanded by all jobs with releases at or after t and effective deadlines at or before $t+L$. Each task T' has at most $\lfloor \max(L - T'.c, 0)/T'.p \rfloor$ such jobs in any interval of length L. (When T' is a sporadic task, this bound is tight.) The second term is a generic "blocking" term, denoted $b(\tau, T, L)$, which accounts for all other forms of demand.[2] Since the form of $b(\tau, T, L)$ will be system-specific, we simply assume for now that $b(\tau, T, L)$ is an arbitrary function. Using these two terms leads to the following inequality:

$$demand(\tau, T, I) \leq \sum_{T' \in \tau(L)} \left\lfloor \frac{L - T'.c}{T'.p} \right\rfloor \cdot T'.e + b(\tau, T, L). \tag{4}$$

When τ is serviced by a dedicated uniprocessor, $available(I) = |I| = L$. On the other hand, when τ is serviced by a supertask $\mathcal{S}$, $available(I) = allocated(\mathcal{S}, t, t+L)$. Hence, (3) and (4) imply that a tardiness violation can occur only if the following condition holds for some task T in τ and some $L \geq T.p + T.c$:

$$allocated(\mathcal{S}, t, t+L) < \sum_{T' \in \tau(L)} \left\lfloor \frac{L - T'.c}{T'.p} \right\rfloor \cdot T'.e + b(\tau, T, L). \tag{5}$$

By the contrapositive of (5), τ will be schedulable whenever the condition shown below holds for each task T in τ, every integer t, and every integer $L \geq T.p + T.c$.

$$allocated(\mathcal{S}, t, t+L) \geq \sum_{T' \in \tau(L)} \left\lfloor \frac{L - T'.c}{T'.p} \right\rfloor \cdot T'.e + b(\tau, T, L) \tag{6}$$

To eliminate $allocated(\mathcal{S}, t, t+L)$ from the left-hand side of (6), we can substitute the lower bound provided by Theorem 1, which produces the following condition:

$$\lfloor \mathcal{S}.w \cdot (L - \alpha) - 2\ell \rfloor + 1 \geq \sum_{T' \in \tau(L)} \left\lfloor \frac{L - T'.c}{T'.p} \right\rfloor \cdot T'.e + b(\tau, T, L). \tag{7}$$

Our goal in reweighting is to assign $\mathcal{S}.w$ so that (7) holds for each task T in τ, every integer t, and every integer $L \geq T.p + T.c$. An obvious first step in this process is to determine when this goal is unattainable. Hence, consider letting $\mathcal{S}.w = 1$ and assume that the right-hand side of (7) is minimized, *i.e.*, zero. Simplifying the inequality yields $\lfloor L - \alpha - 2\ell \rfloor + 1 \geq 0$, which is satisfied iff $L \geq \alpha + 2\ell - 1$. It follows that (7) can be satisfied only if (8), shown below, holds, which we assume henceforth.

$$(\forall T' : T' \in \tau : T'.p + T'.c \geq \alpha + 2\ell - 1) \tag{8}$$

To facilitate the selection of $\mathcal{S}.w$, (7) can be solved for $\mathcal{S}.w$, which produces (9). Notice that, in order to

[2] Blocking terms are often needed to account for complex task behaviors and the looseness of analytical bounds. Hence, the "demand" represented by a blocking term does not necessarily represent requests for processor time.

eliminate the floor operator in the left-hand side of (7), a ceiling operator was introduced. This operator will prove to be problematic for reasons that are explained later.

$$\mathcal{S}.w \geq \frac{\left\lceil \sum_{T' \in \tau(L)} \left\lfloor \frac{L - T'.c}{T'.p} \right\rfloor \cdot T'.e + b(\tau, T, L) \right\rceil + 2\ell - 1}{L - \alpha} \tag{9}$$

To simplify our presentation, we let Δ_J denote the right-hand side of (9), as shown below.

$$\Delta_J(\tau, T, L) = \frac{\left\lceil \sum_{T' \in \tau(L)} \left\lfloor \frac{L - T'.c}{T'.p} \right\rfloor \cdot T'.e + b(\tau, T, L) \right\rceil + 2\ell - 1}{L - \alpha} \tag{10}$$

The J subscript signifies that this function determines the total demand of the component-task set by considering the demand of each job. Hence, we say that Δ_J uses a *job-based demand bound*. (We present a second Δ function below that bounds the total demand using an alternative approach.) Using Δ_J, the reweighting condition that must be satisfied can be expressed as follows:

$$\mathcal{S}.w \geq \max\{\ \Delta_J(\tau, T, L) \mid T \in \tau \wedge L \geq T.p + T.c\ \}. \tag{11}$$

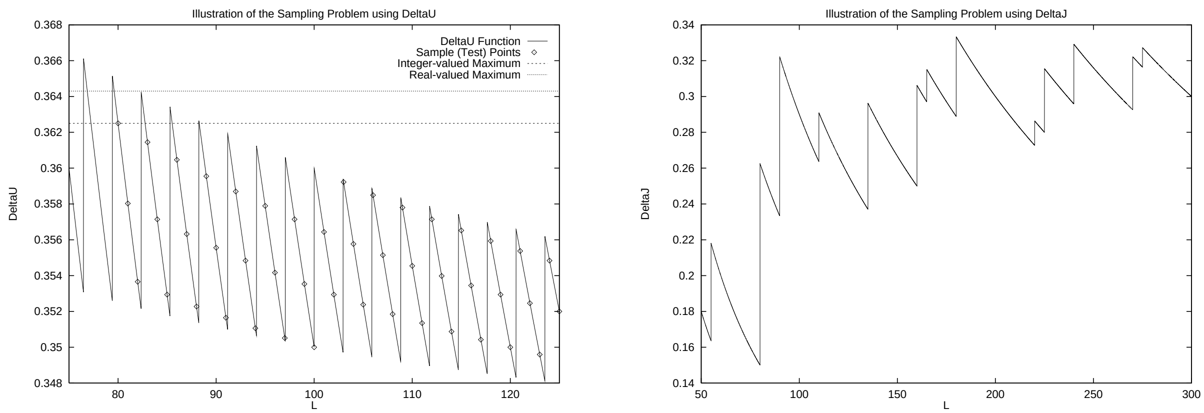
Balancing Schedulability Loss and Computational Overhead. Despite the tightness of the demand bound used in Δ_J (*i.e.*, (4)), it may be preferable to use a looser bound that requires less computation. For instance, consider replacing Δ_J in (11) with Δ_U, which is defined below. Δ_U is derived from Δ_J by simply removing the floor operators from the demand bound. Hence, Δ_U upper bounds Δ_J.

$$\Delta_U(\tau, T, L) = \frac{\left\lceil \sum_{T' \in \tau(L)} T'.u \cdot (L - T'.c) + b(\tau, T, L) \right\rceil + 2\ell - 1}{L - \alpha} \tag{12}$$

Whereas Δ_J uses a job-based demand bound, Δ_U bounds demand by considering the utilization of each task. Hence, we say that Δ_U uses a *utilization-based demand bound*. Δ_U has an advantage over Δ_J in that it requires less time to evaluate than Δ_J since Δ_U's summations can be pre-computed. As we demonstrate later, using Δ_U instead of Δ_J tends to reduce the duration of the reweighting process, but at the expense of precision.

Impact of Sampling. The most direct route to assigning $\mathcal{S}.w$ so that (11) is satisfied would be to determine analytically the value of L that maximizes Δ_J (or Δ_U, depending on which is used) for each task. Although this analysis may seem trivial at first glance, it is not. To understand why, consider a simple case in which τ consists of only three independent, periodic, hard-real-time tasks with parameters (0, 8.8, 80), (0, 7.65, 45), and (0, 3.3, 55). Note that $U = 0.34$. Further, assume that the global scheduler respects strict Pfairness.

Fig. 2**(a)** shows a plot of $\Delta_U(\tau, T, L)$ as a function of L for $T =$(0, 8.8, 80). As shown, the plot does have



(a) **(b)**

Figure 2: **(a)** Plot of Δ_U illustrating the sampling problem posed by the reweighting condition. The points shown on the graph correspond to integer values of L and hence are the points of interest. **(b)** Plot of Δ_J for the same task set. Although the plot is increasing over the interval shown, Δ_J, like Δ_U, will approach its limit from above. (Note that the range of the x-axis differs in these two plots.)

regularly-spaced peaks at which the function is maximized. However, these peaks may not correspond to *integer* values of L. Since (11) only considers integer values of L, using the maximum with respect to real values of L can result in the selection of an overly-pessimistic weight.

The problem that must be addressed is that of sampling a function at regularly-spaced intervals and finding the maximum of the samples. Although this may be relatively straightforward for cases as simple as that shown in Fig. 2**(a)**, it will not be in general. For instance, Fig. 2**(b)** shows the plot of $\Delta_J(\tau, T, L)$ as a function of L for the same component-task set τ. Although the jumps are still predictable, they are no longer uniform. Hence, we do not expect reweighting to be a trivial task, particularly when blocking terms are involved.

Problem Definition. For the remainder of the paper, we assume that the reweighting condition has the following form, where Δ denotes either Δ_J or Δ_U:

$$\mathcal{S}.w \geq \max\{\ \Delta(\tau, T, L) \mid T \in \tau \wedge L \geq T.p + T.c\ \}. \tag{13}$$

Furthermore, we will use w_{opt} to denote the optimal weight assignment with respect to Δ, as shown below.

$$w_{opt} = \max\{\ \Delta(\tau, T, L) \mid T \in \tau \wedge L \geq T.p + T.c\ \}. \tag{14}$$

4 The Algorithm

We now present our reweighting algorithm and briefly explain the concepts underlying its design.

Algorithm Overview. The algorithm, which is shown in Fig. 3, takes the following four input parameters: the task set τ, a search-interval limit λ, a per-task computation limit η, and an initial weight w_0. The latter three arguments provide a means of controlling the duration and precision of the reweighting process. λ (respectively, η) specifies the maximum value of L (respectively, the maximum number of unique values of L) that should be checked for any task. w_0 is then the smallest value of $\mathcal{S}.w$ of interest to the user.

The goal of `Reweight` is to ensure that the variable w satisfies $w \geq \max\{\ \Delta(\tau, T, L) \mid T \in \tau \wedge L \geq T.p{+}T.c\ \}$ when line 17 is executed. This is accomplished by first initializing w to w_0 (line 1) and then checking Δ values in a systematic manner to ensure that w upper bounds each. Whenever a Δ value is found that exceeds w, w is assigned that value (lines 7 and 13). We now give a detailed description of the `Reweight` procedure, followed by a brief discussion of its various mechanisms.

Definitions and Assumptions. We begin by defining the basic elements of `Reweight` and stating assumptions made of each. Since it is not necessary to know the exact form of these elements in order to understand the operation of the algorithm, we do not derive many of them until Sec. 5. First, assume that ϕ is a function that upper bounds Δ for all values of $L \geq L_0$, as formally stated below.

$$(\forall T, L : T \in \tau \wedge L \geq L_0 : \phi(\tau, T, L) \geq \Delta(\tau, T, L)) \tag{15}$$

We refer to L_0 as the *activation point* of ϕ. Furthermore, assume that ϕ changes monotonically as L increases and that the nature of the ϕ's monotonicity is determined by the characteristic function Ψ. Specifically, assume that $\phi(\tau, T, L)$ is decreasing (respectively, non-decreasing) with respect to increasing L when $\Psi(\tau, T) > 0$ (respectively, when $\Psi(\tau, T) \leq 0$). Also, let $w_\phi(\tau, T)$ denote the limit of $\phi(\tau, T, L)$ as $L \to \infty$, as shown below.

$$w_\phi(\tau, T) = \lim_{L \to \infty} \phi(\tau, T, L) \tag{16}$$

Finally, the subroutines `AdvanceSetup` and `Advance` represent a filtering mechanism (explained below) that selects which L values to check. Assume that `AdvanceSetup` is used to initialize the filter and that `Advance` returns the next value of L to check given the last-checked value. In trying to understand the basic operation of the algorithm, it is sufficient to assume that no filtering is performed, *i.e.*, assume that `AdvanceSetup` does nothing and that `Advance` simply returns $L + 1$.

Detailed Description. In line 1, w is initialized. Line 2 then selects the next task T to consider, followed by the initialization of temporary variables in lines 3–5. (n tracks the number of L-values that have been checked for T.) A brute-force check of L-values less than L_0 then occurs in lines 6–9. This loop ensures that $w \geq \max\{\ \Delta(\tau, T, L) \mid L_0 > L \geq T.p + T.c\ \}$ holds. Note that this check is mandatory, *i.e.*, the control parameters η and λ do not affect the duration of the loop. Lines 10–16 then perform the remaining checks.

```
procedure Reweight(τ, λ, η, w_0) returns rational
1:  w := w_0;
2:  for each T ∈ τ do
3:      n := 0;
4:      L := T.p + T.c;
5:      AdvanceSetup(τ, T, L);
6:      while L < L_0 do
7:          w := max(w, Δ(τ, T, L));
8:          n := n + 1;
9:          L := Advance(τ, T, L)
        od;
10:     if Ψ(τ, T) ≤ 0 then
11:         w := max(w, w_φ(τ, T))
        else
12:         while (L < λ) ∧ (n < η) ∧ (w < φ(τ, T, L)) do
13:             w := max(w, Δ(τ, T, L));
14:             n := n + 1;
15:             L := Advance(τ, T, L)
            od;
16:         w := max(w, φ(τ, T, L))
        fi
    od;
17: return w
```

Figure 3: Algorithm for selecting a Pfair supertask weight that satisfies (13).

(The purpose of ϕ, Ψ, and L_0 is explained below.) If ϕ is non-decreasing with increasing L (line 10), then w must upper bound ϕ's limit to ensure that it upper bounds ϕ (and hence Δ), which is guaranteed by line 11. Otherwise, the evaluations continue in lines 12–16 until the termination condition given in line 12 is satisfied. If termination of this loop is *not* caused by either η or λ, then $w \geq \max\{\ \Delta(\tau, T, L) \mid L \geq T.p + T.c\ \}$ holds at line 16. Otherwise, the assignment at line 16 establishes this latter condition. (The correctness of lines 10–16 is explained below.) Finally, the selected w value is returned in line 17 after all tasks have been considered.

Notice that the algorithm never checks whether w exceeds unity. In practice, such a check should be performed to avoid unnecessary computation. We omit this check to simplify the presentation.

Termination Mechanism. We now explain in detail how the monotonicity of ϕ can be exploited both to detect when termination may occur safely and to force premature termination of the algorithm (at the expense of precision). When ϕ monotonically decreases with increasing L, Property (17), shown below, can be used to determine when termination may occur safely.

$$w \geq \phi(\tau, T, \mathcal{L}) \Rightarrow w \geq \max\{\ \phi(\tau, T, L) \mid L \geq \mathcal{L}\ \} \Rightarrow w \geq \max\{\ \Delta(\tau, T, L) \mid L \geq \mathcal{L}\ \} \tag{17}$$

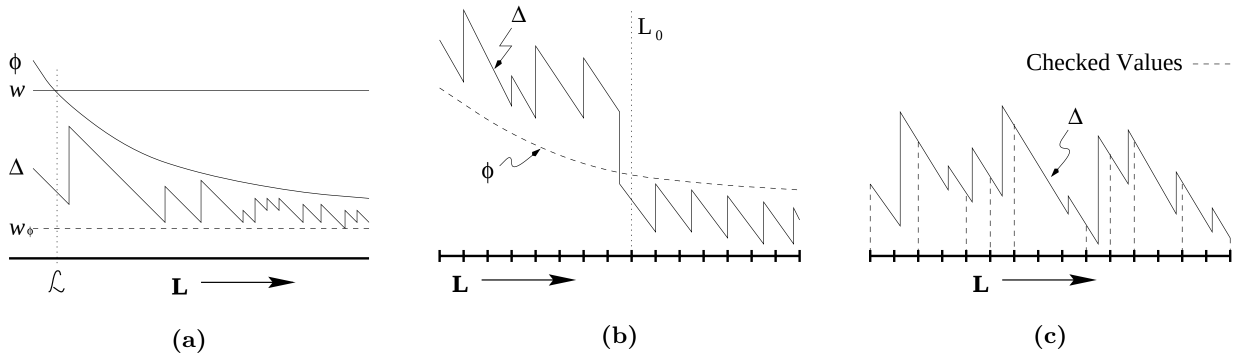


Figure 4: **(a)** Illustration of how ϕ's bounding property can be used to detect when w upper bounds the tail of the Δ function. **(b)** Illustration of how L_0 can be used to exclude transient effects when deriving ϕ. **(c)** Illustration of how the monotonicity of Δ between jumps can be exploited to avoid unnecessary computation. Marks on the lower line correspond to integer values of L, while dashed lines denote which L values are actually checked.

Informally, (17) states that, if w upper bounds ϕ at $L = \mathcal{L}$, then it also upper bounds the tail of the ϕ function. Since (15) implies that the tail of ϕ upper bounds the tail of Δ, it follows that the situation resembles that depicted in Fig. 4**(a)** and that all values of $L \geq \mathcal{L}$ can be skipped. On the other hand, when ϕ is non-decreasing with increasing L, then w must be set to the limit of ϕ as $L \to \infty$ since the limit is the smallest value that upper bounds the tail of ϕ (and hence the tail of Δ). The correctness of lines 10–16 follows.

The purpose of L_0 is to ensure that ϕ tightly bounds Δ. In practice, Δ may experience transient effects due to resource-sharing contention and other complex task interactions. In most cases, such effects appear only when L is small. Consequently, the selection of a ϕ function can be facilitated by setting L_0 so that these effects can be ignored. For instance, consider Fig. 4**(b)**, which illustrates how transient effects can make the selection of a ϕ function difficult. Note that the placement of L_0 shown in the figure guarantees that all transient behavior occurs prior to L_0. Hence, the transient behavior can be ignored when selecting ϕ, as shown. (Recall that a mandatory check of all L-values preceding L_0 is performed by the loop spanning lines 6–9.)

Filtering Mechanism. Consider evaluating $\max\{\ \Delta(\tau, T, L) \mid b \geq L \geq a\ \}$ for some given task T. We can avoid checking every L value in $[a, b]$ by exploiting the fact that both Δ_J and Δ_U monotonically decrease as L increases whenever the ceiling expressions in their numerators remain constant. (See (10) and (12), respectively.) Consequently, only the smallest value of L associated with each unique value of the ceiling expression (greater than its initial value, which is set by $L = a$) must be checked. Here, we are exploiting the predictability of the Δ's jumps, which was noted earlier when considering Fig. 2. Since we can analytically determine where these jumps occur, we can restrict our attention to the first L-value that follows each jump. (Recall that only integer values of L are checked, but that jumps may occur at non-integer points.) Fig. 4**(c)** illustrates this idea. In this figure, marks shown on the lower line denote points at which L is an integer, *i.e.*, points that may require a check, while dashed lines denote where checks are actually performed based on filtering. Notice that only the first integer value of L that follows each jump in the Δ function is checked.

Unfortunately, determining which values of L to check requires exact knowledge of the Δ function's form. Hence, no single algorithm will be applicable in all situations. Due to space limitations, we are unable to present multiple sample implementations. Instead, this optimization is represented abstractly with the `AdvanceSetup` and `Advance` subroutines. For the remainder of the paper, we make the following assumptions concerning `AdvanceSetup` and `Advance`: **(i)** they always terminate, **(ii)** they correctly determine which L-values to check, and **(iii)** they do not alter any of the `Reweight` procedure's variables.

5 Analysis

In this section, we derive expressions for ϕ, Ψ, and w_ϕ, and then state and prove some basic properties of the `Reweight` procedure.

Definitions. First, let $w_\Delta(\tau, T)$ be defined as shown below.

$$w_\Delta(\tau, T) = \lim_{L \to \infty} \Delta(\tau, T, L) \tag{18}$$

Note that (18) parallels (16). Also, note that we are assuming that Δ converges to a single value in the limit.[3] In addition, we will use the shorthand notations given below.[4]

$$w_\Delta(\tau) = \max\{\ w_\Delta(\tau, T) \mid T \in \tau\ \} \tag{19}$$

$$w_\phi(\tau) = \max\{\ w_\phi(\tau, T) \mid T \in \tau\ \} \tag{20}$$

Basic Properties of Reweighting. We state the following properties without proof. (Each of these properties follows trivially from previously-stated observations.)

(P1) $w_{opt} \geq w_\Delta(\tau)$

(P2) If `Reweight` is invoked with $\eta = \lambda = \infty$ (respectively, with $\eta < \infty$ or $\lambda < \infty$) and terminates, then w will be equal to (respectively, at least) $\max(w_{opt}, w_0, \max\{\ w_\phi(\tau, T) \mid T \in \tau \wedge \Psi(\tau, T) \leq 0\})$ upon termination.

By (P1), the smallest weight that can possible satisfy (13) is $w_\Delta(\tau)$. Hence, (P2) implies that initializing w to any value less than $w_\Delta(\tau)$ produces the same result as initializing w to $w_\Delta(\tau)$, assuming that termination occurs in both cases. Based on these observations, we make the following assumption:

(A1) `Reweight` is never invoked with $w_0 < w_\Delta(\tau, T, L)$.

[3]By (10) and (12), Δ fails to converge only if $b(\tau, T, L)/L$ does not converge, which is highly unlikely to occur in practice.

[4]For reasons explained later, it is desirable and should be straightforward to define ϕ so that the relationship $w_\phi(\tau, T) = w_\Delta(\tau, T)$ holds. Hence, $w_\Delta(\tau)$ is expected to always equal $w_\phi(\tau)$.

Deriving ϕ. We (necessarily) begin by deriving a expression for ϕ that will satisfy the assumptions made in the previous section. Recall that ϕ must satisfy two properties. First, ϕ must upper bound Δ for all $L \geq L_0$, as formally stated in (15). Second, ϕ must be monotonic with respect to changes in L. When considering either Δ_J or Δ_U (see (10) and (12)), one immediately unappealing characteristic is the $\tau(L)$ expression in the summation range. $\tau(L)$ is problematic because it introduces jumps into the Δ function that, if duplicated in ϕ, will likely violate the monotonicity requirement. We avoid this problem by imposing the following lower bound on L_0:

$$L_0 \geq \max(\max\{\ T.c \mid T \in \tau\ \},\ \min\{\ T.p + T.c \mid T \in \tau\ \}). \tag{21}$$

The first subexpression in the right-hand side of (21) guarantees that $\tau(L) = \tau$ for all $L \geq L_0$, while the second subexpression is a trivial lower bound for L_0 that is implied by (13). Given (21), we can bound Δ using a function of the form shown below, which is derived directly from (12).

$$\frac{U \cdot L\ -\ \sum_{T' \in \tau} T'.u \cdot T'.c + b(\tau, T, L) + 2\ell}{L - \alpha} \tag{22}$$

Unfortunately, the above function still may not satisfy the monotonicity requirement due to the $b(\tau, T, L)$ term. Indeed, no function can be guaranteed to upper bound $b(\tau, T, L)$ without placing some restriction on its form. Henceforth, we assume that, for all $L \geq L_0$, $b(\tau, T, L)$ is upper bounded by the linear expression defined below.

$$(\forall T, L : T \in \tau \wedge L \geq L_0 : b_1(\tau, T)L + b_2(\tau, T) \geq b(\tau, T, L)) \tag{23}$$

(We chose to use a linear expression because all blocking terms of which we are aware can be tightly bounded by such an expression.) Combining (23) with (22) and reorganizing yields the definitions for ϕ, Ψ, and w_ϕ shown below and Theorem 2.

$$\phi(\tau, T, L) = w_\phi(\tau, T) + \frac{\Psi(\tau,T)}{L\ -\ \alpha} \tag{24}$$

$$\Psi(\tau, T) = b_2(\tau, T) + 2\ell + \alpha \cdot w_\phi(\tau, T) - \sum_{T' \in \tau} T'.u \cdot T'.c \tag{25}$$

$$w_\phi(\tau, T) = U + b_1(\tau, T) \tag{26}$$

Theorem 2 *Given* (8), (21), (24), (25), *and* (26), ϕ *is* **(i)** *monotonically decreasing when* $\Psi > 0$, **(ii)** *constant when* $\Psi = 0$, *and* **(iii)** *monotonically increasing when* $\Psi < 0$, *with respect to increasing* $L \geq L_0$.

Proof. By (8), $L > \alpha + 2\ell - 1$ for all $L \geq \min\{\ T.p + T.c \mid T \in \tau\ \}$. Since $\ell \geq 1 \Rightarrow \alpha + 2\ell - 1 > \alpha$ and (21) implies that $L_0 \geq \min\{\ T.p + T.c \mid T \in \tau\ \}$ holds, it follows that $L - \alpha > 0$ for all $L \geq L_0$. The theorem then follows trivially from (24), (25), and (26). □

Guaranteeing Termination. Given the above definitions, we now turn our attention to the problem of guaranteeing the termination of our reweighting algorithm. In this subsection, we derive necessary and sufficient conditions for non-terminating behavior and then use these conditions to develop a sufficient termination condition. We begin by briefly explaining why non-terminating behavior can occur in `Reweight`, but not in the reweighting algorithm given by us in earlier work [3].

The root cause of non-terminating behavior in the `Reweight` procedure is the form of the demand bound given in (4), which differs from the form of the demand bound used in [3]. When tasks are independent and have integer execution costs, as assumed in [3], the right-hand side of (4) is integer-valued. As a result, (7) can be solved for $\mathcal{S}.w$ without introducing a ceiling operator, *i.e.*, the ceiling operator can be removed from the right-hand side of (9). To illustrate the impact of the ceiling operator, consider the case in which τ consists of only independent, hard-real-time tasks with integer execution costs and in which the global scheduler respects strict Pfairness. Removing the ceiling operator from Δ_J and Δ_U and substituting $\alpha = 0$, $\ell = 1$, $T.c = 0$, and $b(\tau, T, L) = b_1(\tau, T) = b_2(\tau, T) = 0$ yields the function definitions below.

$$\Delta_J(\tau, T, L) = \sum_{T' \in \tau} \left\lfloor \frac{L}{T'.p} \right\rfloor \frac{T'.e}{L} + \frac{1}{L} \qquad\qquad \Delta_U(\tau, T, L) = \frac{1}{L} \lfloor U \cdot L \rfloor + \frac{1}{L}$$

Hence, letting $\phi(\tau, T, L) = U + 1/L$ is sufficient to ensure that the assumptions made of ϕ hold. In addition, this special-case definition bounds the Δ functions more tightly than the general-case definition given in (24). (Applying (24) results in the definition $\phi(\tau, T, L) = U + 2/L$.) A side effect of this improved tightness is that both Δ functions intersect ϕ at the hyperperiod of τ, which guarantees that the exit condition in line 12 holds at the hyperperiod. Hence, the `Reweight` algorithm is guaranteed to terminate when this special-case ϕ definition is used. (This property is formally proven in [3].)

In contrast, using the ϕ definition given in (24) guarantees that ϕ *never* intersects either Δ function. This is because ϕ is derived from Δ_U by using a strict upper bound of the form $\lceil x \rceil < x + 1$ to eliminate the ceiling operator in Δ_U. Consequently, termination is not guaranteed when using the general-case definition of ϕ.

The following theorem gives a necessary and sufficient condition for non-terminating behavior.

Theorem 3 *Given that $\lambda = \eta = \infty$ and $L_0 < \infty$, the* `Reweight` *procedure will not terminate if and only if there exists a task T such that* **(i)** $\Psi(\tau, T) > 0$*,* **(ii)** $\max\{\ \Delta(\tau, T, L) \mid L \geq T.p + T.c\ \} \leq w_\phi(\tau, T)$*, and* **(iii)** $w \leq w_\phi(\tau, T)$ *holds when T is considered at line 2.*

Proof. It follows from Fig. 3 that non-terminating behavior can only be caused by executing the loop spanning lines 12–15 unboundedly. Hence, the necessity of Condition (i) is obvious since this loop would be unreachable otherwise. Furthermore, the combination of Conditions (ii) and (iii) imply that $w \leq w_\phi(\tau, T)$ is invariant across all iterations of the loop. Specifically, Condition (iii) ensures that the invariant holds initially, while the Condition (ii) ensures that the invariant cannot be violated by the assignments in lines 7 and 13.

Consider the implications of this invariant. By Condition (i), Theorem 2, and (24), ϕ monotonically decreases across iterations of the loop spanning lines 12–15 and approaches $w_\phi(\tau, T)$ in the limit. Furthermore, (24) implies that $\phi(\tau, T, L) > w_\phi(\tau, T)$ for all $L > 0$ when Condition (i) holds. It follows from the above invariant that $\phi(\tau, T, L) > w$ is invariant throughout the execution of this loop. Hence, the exit condition in line 12 will never be satisfied when Conditions (ii) and (iii) hold.

It remains only to show that termination is guaranteed when either Condition (ii) or (iii) does not hold. It is trivial to show that $w > w_\phi(\tau, T)$ will eventually hold if either of these conditions does not. Since w is non-decreasing across the execution of the reweighting algorithm, it follows that $w \leq w_\phi(\tau, T)$ cannot be re-established once it has been falsified. Furthermore, the fact that $\phi(\tau, T, L)$ approaches $w_\phi(\tau, T)$ in the limit (see (24)) implies that $\phi(\tau, T, L)$ will eventually grow smaller than any w value that satisfies $w > w_\phi(\tau, T)$. When this occurs, the exit condition in line 12 will be satisfied. Hence, the loop spanning lines 12–15 will eventually terminate if either Condition (ii) or (iii) does not hold. □

Corollary 1, shown below, provides a sufficient termination condition for the `Reweight` procedure.

Corollary 1 *The* `Reweight` *procedure will always terminate when invoked with* $w_0 > w_\phi(\tau)$.

Proof. When the stated condition holds, w will be initialized to a value greater than each $w_\phi(\tau, T)$ value. Hence, the conditions for non-terminating behavior set forth in Theorem 3 cannot be satisfied. □

Corollary 1 characterizes the consequence of introducing the ceiling operator when deriving (9). Effectively, the inclusion of this operator results in a "blindspot" in the algorithm, consisting of the weight range $[w_\Delta(\tau), w_\phi(\tau)]$. (Note that (15) implies that $w_\Delta(\tau) \leq w_\phi(\tau)$.) This range follows from the combination of (P1) and Corollary 1. Specifically, (P1) bounds the range of all possible solutions to $[w_\Delta(\tau), 1]$, while Corollary 1 implies that `Reweight` should only be used to search for solutions in the range $(w_\phi(\tau), 1]$. However, it is important to note that when $w_\Delta(\tau) = w_\phi(\tau)$, this range is reduced to a single point. In such cases, this limitation will be of very little practical importance since `Reweight` will be capable of assigning weights that are arbitrarily close to $w_\Delta(\tau)$. Furthermore, observe that the relationship $w_\Delta(\tau) < w_\phi(\tau)$ can only arise when ϕ bounds Δ loosely. Given the flexibility provided by L_0, it is doubtful that such loose bounds will ever be used. Hence, this limitation, though theoretically unappealing, should seldom, if ever, cause problems in practice.

Bounding the Number of Function Evaluations. We now focus on deriving a bound on the number of Δ and ϕ evaluations performed when `Reweight` is invoked. Although a derivation of time complexity figures would be preferable, such a derivation is not possible due to the system-dependent forms of the filtering subroutines, `AdvanceSetup` and `Advance`, and the blocking terms. The following lemma and theorem state and prove pessimistic bounds on the number of Δ and ϕ evaluations performed. Specifically, the presented bounds are derived by assuming that filtering is not used and that `Reweight` is invoked with $\lambda = \eta = \infty$.

Lemma 1 *Suppose $w = \hat{w} > w_\phi(\tau, T)$ holds when some task T is considered in line 2 of the* `Reweight` *procedure. If $\Psi(\tau, T) \leq 0$, then Δ is evaluated at most $\max(L_0 - T.p - T.c,\ 0)$ times and ϕ is never evaluated when considering T. If $\Psi(\tau, T) > 0$, then Δ is evaluated at most $\max(L_0 - T.p - T.c,\ 0) + \max(L^* - \max(T.p + T.c,\ L_0),\ 0)$ times and ϕ is evaluated at most $\max(L^* - \max(T.p + T.c, L_0),\ 0) + 1$ times when considering T, where $L^* = \left\lceil \frac{\Psi(\tau,T)}{w_0 - w_\phi(\tau,T)} + \alpha \right\rceil$.*

Proof. L-values are checked in three locations within the `Reweight` procedure. First, lines 6–9 evaluate Δ for each selected L in $[T.p + T.c,\ \max(T.p + T.c,\ L_0))$. Hence, Δ is evaluated at most $\max(L_0 - T.p - T.c,\ 0)$ times by these lines. In addition, these are the only L-values checked when $\Psi(\tau, T) \leq 0$.

Second, lines 12–15 evaluate both Δ and ϕ once per loop iteration and hence once for each selected L in $[\max(T.p + T.c, L_0),\ \max(L^*,\ \max(T.p + T.c,\ L_0)))$, where L^* is the smallest L that satisfies $w \geq \phi(\tau, T, L)$. It follows that at most $\max(L^* - \max(T.p + T.c,\ L_0),\ 0)$ evaluations occur due to iterations of this loop. Since w is non-decreasing and ϕ monotonically decreases across iterations of the loop, changing (increasing) w can only reduce the number of loop iterations performed prior to termination. Therefore, the number of iterations is maximized when w remains constant, *i.e.*, $w = \hat{w}$, throughout the loop's execution. Solving $\hat{w} \geq \phi(\tau, T, L)$ for L^* yields $L^* = \lceil \Psi(\tau, T)/(\hat{w} - w_\phi(\tau, T)) + \alpha \rceil$.

Finally, ϕ is evaluated two additional times, at lines 12 and 16, for the L-value that causes the loop to exit. However, since these evaluations use the same parameters, the evaluation in line 16 can be avoided by using a temporary variable. Hence, we count both as a single evaluation. The lemma follows. □

Theorem 4 *If* `Reweight` *is invoked with some w_0 satisfying $w_0 > w_\phi(\tau)$, then Δ is evaluated at most*

$$\sum_{T \in \tau} \max(L_0 - T.p - T.c,\ 0) + \sum_{T \in \tau \wedge \Psi(\tau,T) > 0} \max(L^* - \max(T.p + T.c,\ L_0),\ 0)$$

times and ϕ is evaluated at most

$$\sum_{T \in \tau \wedge \Psi(\tau,T) > 0} [\max(L^* - \max(T.p + T.c,\ L_0),\ 0) + 1]$$

times, where $L^* = \left\lceil \frac{\Psi(\tau,T)}{w_0 - w_\phi(\tau,T)} + \alpha \right\rceil$.

Proof. The theorem follows easily from the previous lemma. □

6 Experimental Results

In this section, we summarize the results of two series of experiments. These experiments were conducted to determine the efficacy of using supertasks to reduce the schedulability loss caused by the integer-cost problem and by task synchronization.

Integer-cost Problem. In the first series of experiments, we randomly generated 1,000 component-task sets, each of which consisted of $5x$ independent, hard-real-time tasks, for each $x \in [1, 16]$. For each set, the schedulability loss relative to U was computed for **(i)** using Pfair scheduling without supertasks, **(ii)** using the old reweighting algorithm [3] with each of Δ_J and Δ_U, and **(iii)** using the new reweighting algorithm (of this paper) with each of Δ_J and Δ_U. Recall that the integer-cost assumption of Pfair scheduling, which is also an assumption of the old reweighting algorithm, forces tasks to reserve processor time in multiples of the scheduling quantum. Hence, using the old reweighting algorithm results in schedulability loss due to *both* the integer-cost assumption and reweighting.

The schedulability-loss sample means are shown in Fig. 5**(a)**. In addition, Fig. 5**(b)** shows the associated relative error bounds based upon the half-length of a 99% confidence interval. Due to the extremely low schedulability loss resulting from reweighting with Δ_J, the line corresponding to the use of Δ_J with the old reweighting algorithm has been omitted for clarity: the omitted line is virtually co-linear with that of the "Pfair w/o Supertasks" case. Note also that the line corresponding to the use of Δ_J with the *new* reweighting algorithm is virtually co-linear with the x-axis, *i.e.*, *almost no schedulability loss occurs in this case.*[5]

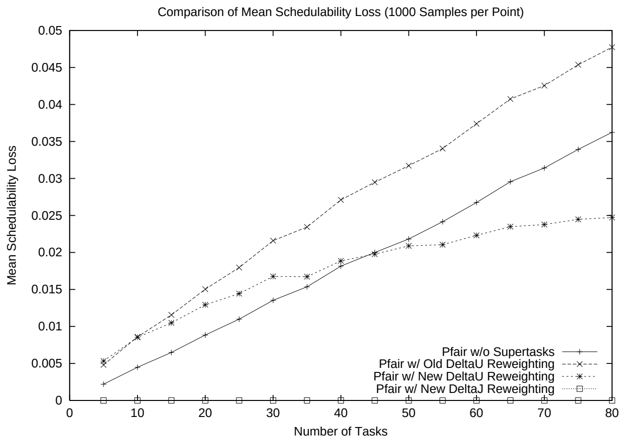
Although the Δ_U version of our new reweighting algorithm does not perform as well as the Δ_J version, Fig. 5**(a)** shows that the mean schedulability loss observed when using the new reweighting algorithm with Δ_U improved upon that of Pfair scheduling without supertasks for sets consisting of at least 45 tasks. Furthermore, Fig. 5**(c)** shows that the majority of sets consisting of 35 tasks or more experienced less loss when using new reweighting algorithm with Δ_U than when using Pfair scheduling without supertasks.

Although the Δ_J version of our new reweighting algorithm is the clear choice in terms of minimizing schedulability loss, its precision is obtained at the expense of computational overhead. The cost of reweighting with Δ_J is illustrated in Fig. 5**(d)**. Specifically, this graph shows that the use of Δ_J with the new reweighting algorithm instead of Δ_U resulted in approximately 10,000 times the number of function evaluations.

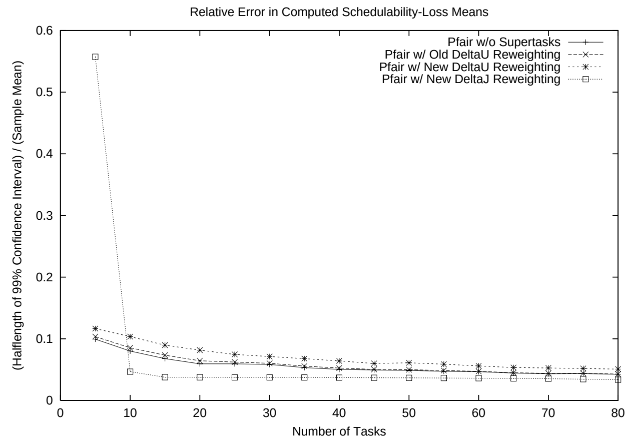
The Synchronization Problem. In the second series of experiments, we considered the problem of scheduling tasks that share resources through semaphore-based locks. Specifically, we compared the use of the hybrid SP/SWSP protocol[6] proposed in [4], which supports task synchronization at the global-scheduling level, to the use of the priority-ceiling protocol (PCP) [6] within a supertask. (See [6] for details on deriving blocking terms when using the PCP.)

[5]The high relative error around $N = 5$ in Fig. 5**(b)** is likely due to the combination of the extremely low schedulability loss and the inherent variability encountered when using a small number of tasks. This follows from the fact that relative error is computed by dividing the absolute error by the computed mean. Hence, when the mean is very small, even slight variations in the samples result in very large relative-error estimates.

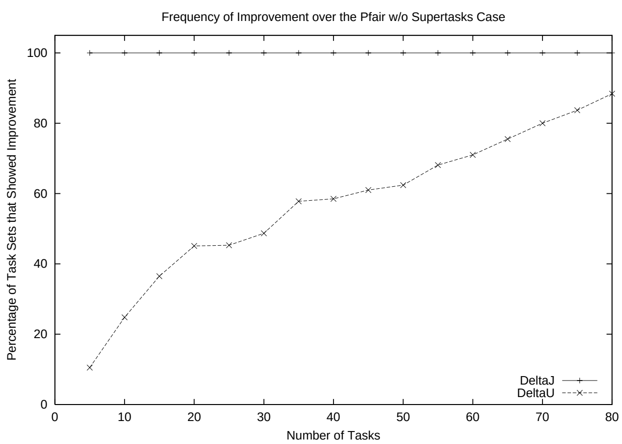
[6]The SP/SWSP protocol is actually a combination of two protocols. When critical sections are sufficiently short, the skip protocol (SP) is use to prevent critical sections from being preempted. On the other hand, long critical sections are executed remotely by special server tasks under the static-weight server protocol (SWSP). These protocols are explained in detail in [4].



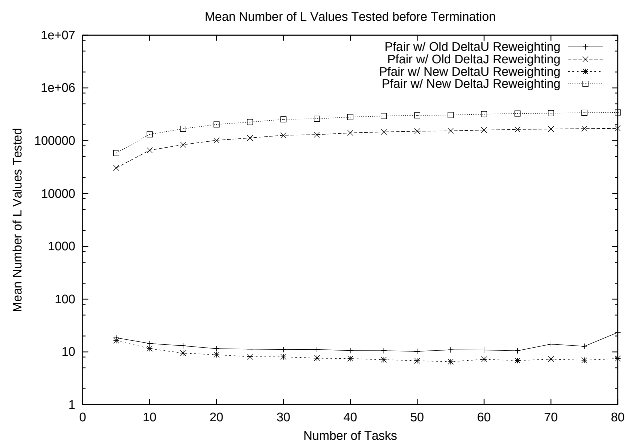
(a) Mean Schedulability Loss



(b) Relative Error



(c) Improvement Frequency



(d) Number of Computations

Figure 5: Results from the first series of experiments, including **(a)** the mean schedulability loss, **(b)** the relative error based upon the half-length of a 99% confidence interval, **(c)** the percentage of task sets tested that experienced more schedulability loss due to the Pfair's integer-cost requirement than from reweighting, and **(d)** the mean number of Δ evaluations performed by the reweighting algorithm before terminating (shown in log scale). Each point is based upon the performance of 1,000 randomly-generated task sets.

To compare these two alternatives, we conducted four experiments, in which critical-section durations were limited to 1/4, 1, 2, and 4 quanta, respectively. The latter values were chosen because the hybrid SP/SWSP protocol has been shown to perform well when critical-section durations are at most one-quarter of the quantum size [4]. However, its performance drops off quickly as critical-section durations increase.

For each duration bound, the schedulability loss experienced by 15,000 randomly-generated sets of hard real-time tasks was computed. As in the first series of experiments, we determined the schedulability loss due to using each of Δ_U and Δ_J with the new reweighting algorithm. Fig. 6 shows the (smoothed) sample data collected from these experiments. (Due to non-uniform sample distribution, no sample means were computed.) In each graph, schedulability loss is plotted against the task set's cumulative locking utilization, which is the

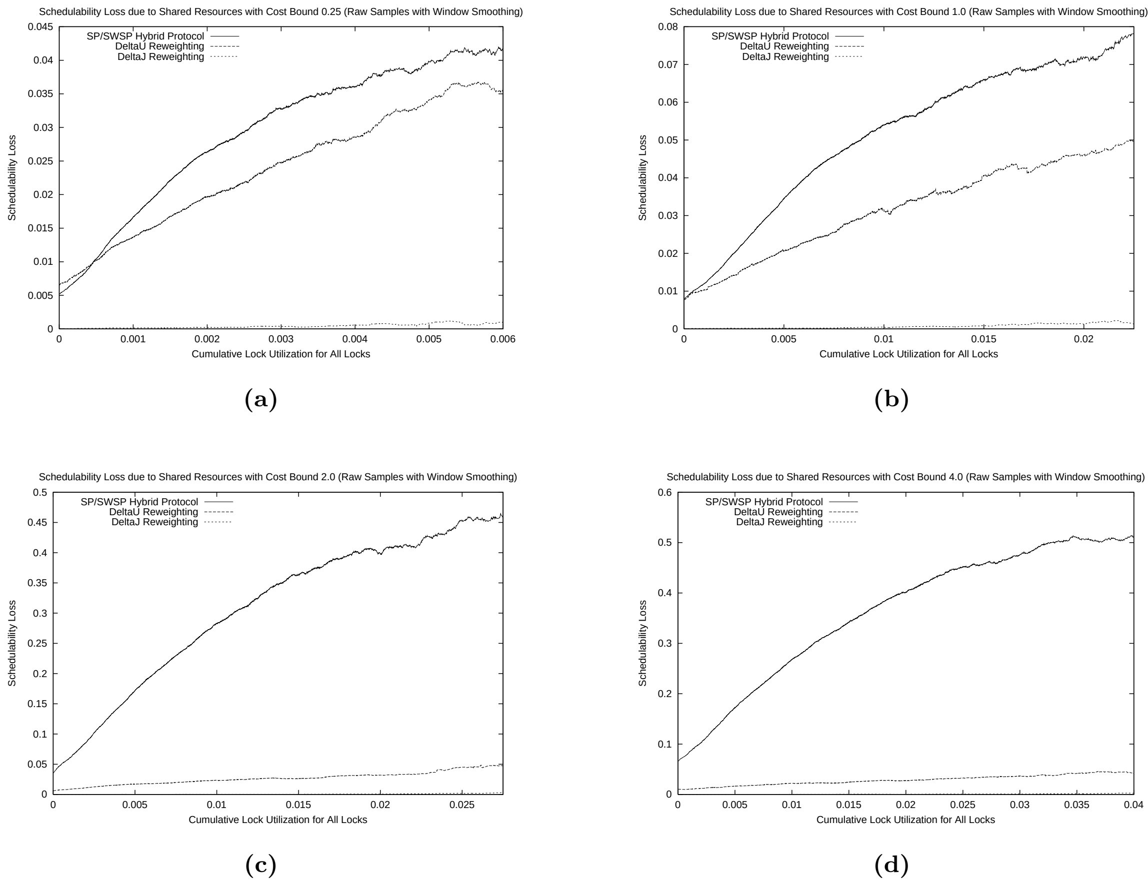


Figure 6: Results for second series of experiments. Due to non-uniform sample distribution, raw sample data is shown. (The sample data has been smoothed to make the performance trend easier to see.) The graphs show the schedulability loss due to resource sharing when using supertasks with the priority-ceiling protocol and when using Pfair scheduling directly with the SP/SWSP protocol. For the former case, schedulability loss is shown for both the Δ_J and Δ_U cases. In these experiments, critical-section durations were limited to at most **(a)** 1/4, **(b)** 1, **(c)** 2, and **(d)** 4 quanta.

fraction of U that is associated with critical sections. The small range of the x-axis is primarily an artifact of the random-generation process. However, the schedulability of a system degrades quickly as locking utilization is increased due to the disruptive nature of locking. In particular, if a critical section is preempted, then its length can effectively be magnified by several orders of magnitude.

As shown, reweighting with Δ_J again performs extremely well, resulting in very low schedulability loss in all cases. Indeed, the associated line is difficult to see in the figure because it is only slightly above the x-axis. On the other hand, reweighting based upon Δ_U and the hybrid SP/SWSP protocol perform comparably for 1/4-bound case. However, reweighting with Δ_U clearly improves upon the hybrid protocol in all other cases.

Although these results clearly suggest that supporting task synchronization through the use of supertasks is preferable, it is important to keep in mind that this approach is only applicable in special cases. Specifically, the PCP can be used only when all tasks that require the lock in question are component tasks of the same supertask. On the other hand, the use of hybrid protocol is unrestricted.

7 Conclusion

In this paper, we have presented and formally analyzed a new reweighting algorithm that extends prior work in three important ways. First, tasks are permitted to use non-integer execution costs. As a result, supertasking can be used to reduce schedulability loss due to the integer-cost requirement of Pfair scheduling. Second, blocking terms have been incorporated into the analysis. Since blocking terms are used to account for a wide range of behaviors commonly found in actual systems, including non-preemptivity, resource sharing, and deferrable execution, their inclusion is quite important from a practical standpoint. Finally, the model of the global scheduler has been generalized to permit the use of supertasks in a wider range of systems.

We also conducted a series of experiments to determine the efficacy of using supertasking to reduce schedulability loss caused by the integer-cost requirement and by task synchronization. The results of these experiments suggest not only that supertasking is an effective mechanism for reducing schedulability loss, but also that reweighting with Δ_J may often result in *negligible* loss. This latter result is quite interesting because it suggests that the use of supertasks may result in near-optimal processor utilization in multiprocessor real-time systems.

References

[1] S. Baruah, N. Cohen, C.G. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[2] A. Chandra, M. Adler, and P. Shenoy. Deadline fair scheduling: Bridging the theory and practice of proportionate fair scheduling in multiprocessor systems. Technical Report TR00-38, Department of Computer Science, University of Massachusetts at Amherst, December 2000.

[3] P. Holman and J. Anderson. Guaranteeing pfair supertasks by reweighting. In *Proc. of the 22nd IEEE Real-time Systems Symposium*, pp. 203–212. December 2001.

[4] P. Holman and J. Anderson. Locking in pfair-scheduled multiprocessor systems. In *Proc. of the 23rd IEEE Real-time Systems Symposium*, pp. 149–158. December 2002.

[5] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In *Proc. of the Twentieth IEEE Real-Time Systems Symposium*, pp. 294–303, December 1999.

[6] R. Rajkumar. *Synchronization in real-time systems – A priority inheritance approach.* Kluwer Academic Publishers, Boston, 1991.

[7] A. Srinivasan, P. Holman, and J. Anderson. The case for fair multiprocessor scheduling. In submission, 2002.

[8] A. Srinivasan, P. Holman, J. Anderson and J. Kaur. Multiprocessor scheduling in processor-based router platforms: issues and ideas. In submission, 2002.