Group-based Pfair Scheduling*

Philip Holman and James H. Anderson

Department of Computer Science

University of North Carolina

Chapel Hill, NC 27599-3175

Phone: (919) 962-1757

Fax: (919) 962-1799

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

August 2005

Abstract

We consider 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 (or group-based) scheduling.

In this paper, we present a generalized framework for "reweighting" supertasks. 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. We consider the use of both fully preemptive and quantum-based scheduling within a supertask.

Keywords: hierarchal, reweighting, supertask, Pfairness, scheduling, multiprocessor

*Work supported by NSF grants CCR 9732916, CCR 9972211, CCR 9988327, ITR 0082866, CCR 0204312, and CCR 0309825. Preliminary versions of some content appeared previously in (Holman and Anderson, 2001, 2003).

1 Introduction

Multiprocessor real-time 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 in real-time systems, 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.* (Baruah et al., 1996). Pfair scheduling is presently the only known optimal method for scheduling recurrent real-time tasks in a multiprocessor system. Under Pfair scheduling, each task is assigned a *weight* that specifies the fraction of a single processor to which that task is entitled. Scheduling decisions are then made so that each task receives approximately its designated share of processor time.

Unfortunately, Pfair scheduling poses many practical problems. First, migration is unrestricted. Even if tasks do not need to execute on specific processors, unrestricted migration can result in significant overhead (Moir and Ramamurthy, 1999). Second, task suspensions can result in wasted processor time in the form of partially used quanta. Minimizing such waste requires the use of shorter quanta, which increases scheduling overhead and makes efficient synchronization more difficult (Holman, 2004). Finally, Pfair scheduling is somewhat strict in that each task is required to make progress at an approximately steady rate. As a result, the scheduler tends to evenly distribute each task's quanta over time, which is not desirable in cache-based systems.

One technique that has the potential to ameliorate these problems is the use of group-based, or *hierarchal*, scheduling techniques (Holman, 2004; Holman and Anderson, 2001, 2002a,b, 2003; Moir and Ramamurthy, 1999). Under this approach, task *groups* are scheduled instead of individual tasks; when a group is selected to execute, an internal scheduler is invoked to distribute the processor time among the group members. Using the terminology of Moir and Ramamurthy (Moir and Ramamurthy, 1999), a Pfair-scheduled group is called a *supertask*, and a supertask member is called a *component task*. Supertasking effectively relaxes the strictness of Pfair scheduling: the group is required to make progress at a steady rate rather than individual tasks. Unfortunately, Moir and Ramamurthy demonstrated that using an ideal weight assignment with a supertask cannot, in general, guarantee the timeliness of its component tasks.

Contributions of this Paper. In this paper, we extend the supertasking approach proposed by Moir and Ramamurthy (Moir and Ramamurthy, 1999). We present four primary contributions. First, we show that scheduling within a supertask is analogous to scheduling on a dedicated uniprocessor. Second, we identify the root cause of the timing violations observed in (Moir and Ramamurthy, 1999). Third, we present a flexible framework for selecting a supertask weight that guarantees the timeliness of its component tasks. We demonstrate the utility of this framework by considering two common scenarios. Finally, we compare and contrast Pfair scheduling with supertasks to partitioning. As we later explain, the two approaches are quite similar conceptually. Because of this, many benefits of partitioning can also be obtained through the use of supertasks.

The remainder of the paper is organized as follows. We begin by summarizing relevant background information in Section 2. Section 3 provides insight into supertasking and its relationship to other approaches. In Section 4, a sufficient schedulability condition for component tasks is derived. In Section 5, this last condition is used to derive a "reweighting" condition, which is a sufficient schedulability condition in the form of a collection of supertask weight restrictions. Section 6 focuses on selecting a "safe" supertask weight, *i.e.*, one that satisfies the reweighting condition and hence guarantees schedulability. Section 7 then presents and proves basic properties of an algorithm for selecting safe supertask weights. Section 8 presents the results of an experimental evaluation of supertasking. We conclude in Section 9.

2 Background

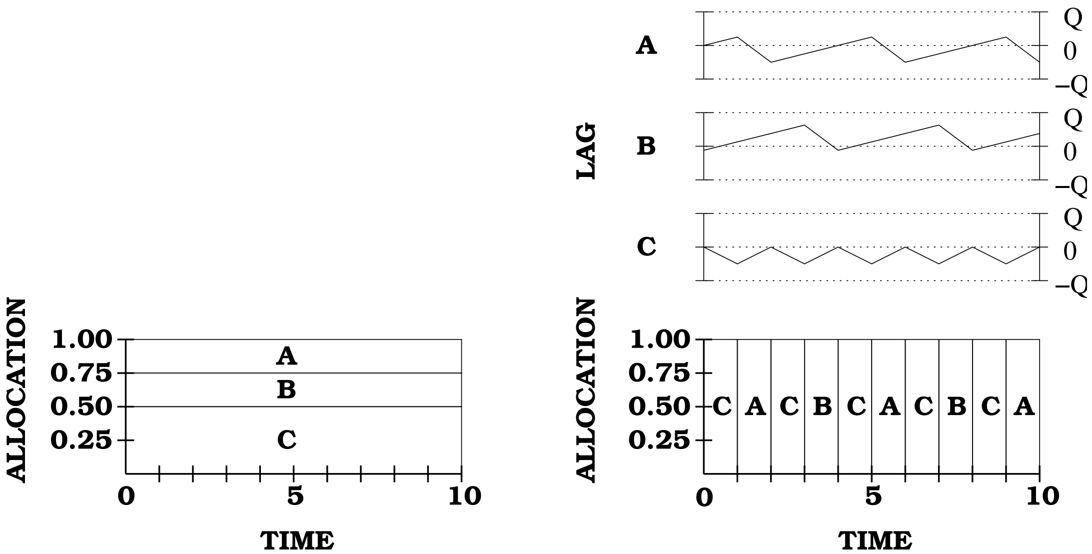
In this section, we summarize background information that is related to the results presented herein.

The problem. We consider the scheduling of a collection τ of component tasks within the processor time allocated to the supertask that represents them. We let $\mathcal{S}$ denote both the supertask and the set of component tasks. The supertask is assume to be scheduled as a Pfair task (explained below), while the scheduling of component tasks will vary based on the scenario under consideration.

Pfair scheduling. Under Pfair scheduling, each task T is characterized by a *weight* $T.w$ in the range $(0, 1]$. Conceptually, $T.w$ is the fraction of a single processor to which T is entitled. We let $T = \mathbf{PF}(w)$ to denote a Pfair task with $T.w = w$.

Time is subdivided into a sequence of fixed-length *slots*. To simplify the presentation, we use the slot length as the basic time unit, *i.e.*, slot i corresponds to the time interval $[i, i + 1)$. Within each slot, each processor may be allocated to at most one task. For instance, in Figure 1(b), task B is scheduled in slot 3, which corresponds to the time interval $[3, 4)$. (The rest of this figure is considered in detail below.) Task migration is allowed. We let Q denote the *quantum* size, *i.e.*, the amount of processor time actually provided by each processor within each slot. In a real system, some processor time is unavoidably consumed in each slot by system activities, such as scheduling. We refer to such overhead as *per-slot* overhead. When practical overheads are ignored, as is commonly done in the literature, $Q = 1$.

Pfair scheduling tracks the allocation of processor time in a fluid schedule; deviation is formally expressed



(a) Fluid scheduler **(b) Lag-based scheduler**

Figure 1: Sample schedules for $\tau = \{A, B, C\}$ where $A = \mathbf{PF}\left(\frac{1}{4}\right)$, $B = \mathbf{PF}\left(\frac{1}{4}\right)$, and $C = \mathbf{PF}\left(\frac{1}{2}\right)$. **(a)** Schedule produced by a fluid scheduler. **(b)** Schedule produced by a Pfair lag-based scheduler.

as $lag(T, t)$, which is defined below.

$$lag(T, t) = fluid(T, 0, t) - received(T, 0, t) \tag{1}$$

In the above equation, $received(T, t_1, t_2)$ denotes the amount of processor time received by T over $[t_1, t_2)$, while $fluid(T, t_1, t_2)$ denotes the amount of processor time guaranteed by fluid scheduling over this interval. As explained in (Holman, 2004), $fluid(T, t_1, t_2)$ is defined as shown below.[1]

$$fluid(T, t_1, t_2) = T.w \cdot (t_2 - t_1) \cdot Q \tag{2}$$

The above formula follows from the fact that each processor provides $(t_2 - t_1) \cdot Q$ units of processor time to tasks over $[t_1, t_2)$. Each task T is then entitled to a fraction $T.w$ of this quantity. (See (Holman, 2004) for a more detailed explanation of fluid scheduling.) Using this notion of lag, the *Pfairness* timing constraint for a task T can be formally defined as shown below.

$$\text{for all } t,\ |lag(T, t)| < Q \tag{3}$$

Informally, T's allocation must always be within one quantum of its fluid allocation.

Figure 1(a) shows ideal (*i.e.*, $Q = 1$) fluid and Pfair uniprocessor schedules for a task set containing three Pfair tasks: $A = \mathbf{PF}\left(\frac{1}{4}\right)$, $B = \mathbf{PF}\left(\frac{1}{4}\right)$, and $C = \mathbf{PF}\left(\frac{1}{2}\right)$. In Figure 1(b), changes in each task's lag are shown

[1]Because $Q = 1$ is commonly assumed, Q typically does not appear in similar formulas in the literature.

across the top of the schedule.

Baruah *et al.* (Baruah et al., 1996) showed that a schedule satisfying (3) exists on M processors for a set τ of Pfair tasks if and only if the following condition holds.

$$\sum_{T \in \tau} T.w \leq M \tag{4}$$

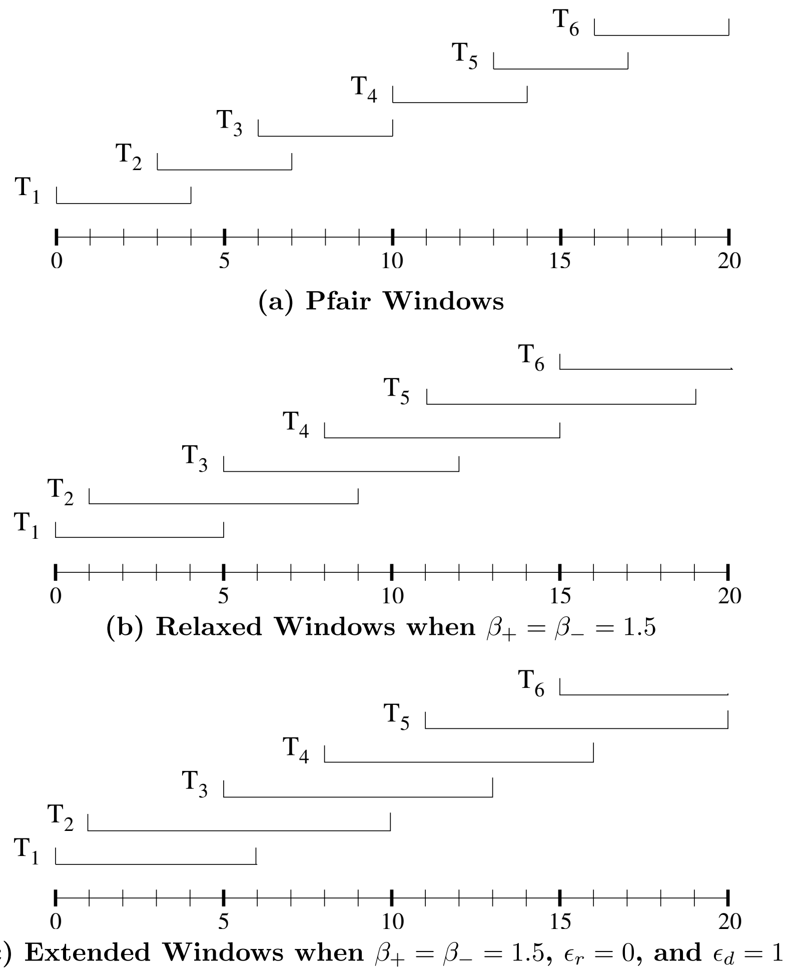
Subtasks and windows. The use of quantum-based scheduling effectively subdivides each task into a sequence of quantum-length *subtasks*. Scheduling constraints, *e.g.*, (3), have the effect of specifying a *window* of slots in which each subtask must be scheduled. We let T_i denote the i^{th} subtask of task T, and let $\omega(T_i)$ denote the window of that subtask. Figure 2(a) shows the window within which each subtask of the task $\mathbf{PF}\left(\frac{3}{10}\right)$ must execute based on (3). For example, $\omega(T_2) = [3, 7)$. $\omega(T_i)$ extends from T_i's *pseudo-release*,[2] denoted $r(T_i)$, to its *pseudo-deadline*, denoted $d(T_i)$. In Figure 2(a), $r(T_2) = 3$ and $d(T_2) = 7$. A schedule satisfies Pfairness if and only if each subtask T_i executes in the interval $[r(T_i), d(T_i))$.

Pfair schedulers. Several Pfair algorithms have been proposed, including PF (Baruah et al., 1996), PD (Baruah et al., 1995), PD^2 (Anderson and Srinivasan, 2001), and EPDF (Anderson and Srinivasan, 2000; Srinivasan and Anderson, 2003). Each of PF, PD, and PD^2 is optimal, *i.e.*, its use will result in a Pfair schedule whenever (4) is satisfied. EPDF has been shown to be optimal only for systems of at most two processors (Anderson and Srinivasan, 2000). Despite this, EPDF offers some practical advantages over the optimal algorithms, such as lower scheduling overhead.

In this paper, we consider only the *guarantees* provided by the scheduler and base our work on properties that follow from these guarantees. There are two primary benefits to abstracting the scheduler in this way. First, our results can be applied easily to both the optimal and sub-optimal Pfair schedulers. As demonstrated by Anderson and Srinivasan (Anderson and Srinivasan, 2000), sub-optimal policies, such as EPDF, are capable of providing fairness guarantees similar to, but weaker than, the Pfairness guarantee. Such relaxed fairness poses an interesting trade-off since weaker guarantees are often offset by practical gains, such as lower scheduling overhead. By enabling the use of our results under a variety of schedulers, we lay the foundation for a quantitative evaluation of this trade-off. Second, more scheduling policies will likely be proposed in the future. By developing a model for Pfair-like schedulers, we provide some forward compatibility with future work and try to avoid the need to revisit this issue each time a new scheduler is proposed.

To characterize these guarantees, we use a four-parameter model, previously proposed in (Holman and Anderson, 2003). First, we let β_- (≥ 1) and β_+ (≥ 1) denote (real-valued) lower and upper *lag scalers*. These scalers are multiplied by $-Q$ and Q, respectively, to yield the actual lag bounds guaranteed by the

[2]The "pseudo" prefix avoids confusion with job releases and deadlines. This prefix will be omitted when the proper interpretation is clearly implied.



(a) Pfair Windows

(b) Relaxed Windows when $\beta_+ = \beta_- = 1.5$

(c) Extended Windows when $\beta_+ = \beta_- = 1.5$**,** $\epsilon_r = 0$**, and** $\epsilon_d = 1$

Figure 2: The first six windows of a task with weight $\frac{3}{10}$ are shown up to time 20. **(a)** Windows defined by Pfairness constraint. **(b)** Relaxed windows defined by $\beta_+ = \beta_- = 1.5$. **(c)** Extended windows defined by $\beta_+ = \beta_- = 1.5$, $\epsilon_r = 0$, and $\epsilon_d = 1$.

scheduler, as shown below.

$$\text{for all } t, \; -Q \cdot \beta_- < \mathit{lag}(T, t) < Q \cdot \beta_+ \tag{5}$$

To simplify the presentation, we let

$$\beta = \beta_+ + \beta_-. \tag{6}$$

The constraint given by (5) generalizes (3), which corresponds to the $\beta_- = \beta_+ = 1$ case.

Relaxing lag bounds scales each subtask window. However, due to the use of quantum-based scheduling, windows are clipped to slot boundaries, resulting in non-uniform scaling. We refer to the windows defined by (5) as *relaxed* windows. Figure 2(b) shows the first six relaxed windows for a task with weight $\frac{3}{10}$ when $\beta_- = \beta_+ = 1.5$; Figure 2(a) shows the corresponding Pfair window layout. Notice that $\omega(T_2)$'s release occurs two slots earlier in Figure 2(b), while $\omega(T_3)$'s release occurs only one slot earlier.

The second parameter pair is ϵ_r and ϵ_d, which denote the number of slots by which each pseudo-release and pseudo-deadline, respectively, is extended (beyond its lag-based placement). More precisely, the scheduler treats a subtask with a relaxed window spanning $[t_r, t_d)$ as having the window $[t_r - \epsilon_r, t_d + \epsilon_d)$. Figure 2(c) shows the window layout obtained by $\beta_- = \beta_+ = 1.5$, $\epsilon_r = 0$, and $\epsilon_d = 1$. Notice that each deadline is extended by one slot, relative to Figure 2(c), due to ϵ_d. Such windows are called *extended* windows. For example, T_2 in Figure 2(c) has an extended deadline at time 10. We let

$$\epsilon = \epsilon_r + \epsilon_d. \tag{7}$$

Basic properties. We now state without proof the following properties of the global scheduling. (The proofs are given in an appendix.) These properties represent only the *guarantee* provided by the scheduler; we make no assumptions about how this guarantee is provided by the scheduler, beyond those already stated.

The theorem, shown below, provides formulas for determining the placement of extended windows.

Theorem 1 *The following formulas define the placement of extended windows:*

$$r(T_i) = \left\lfloor \frac{i - \beta_+}{T.w} \right\rfloor - \epsilon_r \qquad\qquad d(T_i) = \left\lceil \frac{(i-1) + \beta_-}{T.w} \right\rceil + \epsilon_d.$$

The next lemma bounds the number of slots spanned by a sequence of n consecutive windows, which we refer to as an *n-span*. For instance, the interval $[3, 13)$ in Figure 2(a) is a 3-span since $r(T_2) = 3$ and $d(T_4) = 14$. In general, each n-span corresponds to an interval $[r(T_{i+1}), d(T_{i+n}))$ for some integer i.

Lemma 1 *Every sequence of consecutive subtasks $T_{i+1}, \ldots, T_{i+n}$ satisfies the following:*

$$\left\lceil \frac{n + \beta - 2}{T.w} \right\rceil + \epsilon \le d(T_{i+n}) - r(T_{i+1}) \le \left\lceil \frac{n + \beta - 2}{T.w} \right\rceil + \epsilon + 1.$$

Periodic and sporadic tasks. Each periodic (Liu and Layland, 1973) and sporadic (Mok, 1983) task T is characterized by four parameters: an *offset* $T.\phi$, a per-job *execution requirement* $T.e$, a *period* $T.p$, and a *relative deadline* $T.d$. Each time the task is invoked, a *job* is released that must complete within $T.d$ time units. The first invocation occurs at time $T.\phi$. Under the periodic (respectively, sporadic) task model, the next invocation occurs exactly (respectively, at least) $T.p$ time units after the previous invocation. Each job requires $T.e$ units of processor time to complete. We let $T = \mathbf{P}(\phi, e, p, d)$ (respectively, $T = \mathbf{S}(\phi, e, p, d)$) denote a periodic (respectively, sporadic) task T with $T.\phi = \phi$, $T.e = e$, $T.p = p$, and $T.d = d$. We make the simplifying assumption that $T.p = T.d$ for each task T. (Other cases are considered in (Holman, 2004).) Under such an assumption, a task T is often characterized by its *utilization* $T.u$, which is defined by $\frac{T.e}{T.p}$. Informally, a task's utilization is the fraction of a single processor's time that will be consumed by that task in the limit.

3 Understanding Supertasking

Supertasking is a natural extension of Pfair scheduling. Baruah, Cohen, Plaxton, and Varvel Baruah et al. (1996) observed that timeliness can be guaranteed by assigning a dedicated processor to each task. Under such an approach, each processor's speed could be scaled so that it exactly matches the requirements of the assigned task. As a result, processor speeds would vary. Pfair scheduling, by design, *simulates* such a system by time-sharing "virtual" processors (*i.e.*, the Pfair tasks) among a collection of M identical processors. Each Pfair task effectively acts as a dedicated (virtual) processor for the associated task.

Virtual processors differ from dedicated processors in that the amount of processor time available to the task set varies over time. Proper analysis requires that these variations be bounded and predictable. Under Pfair scheduling, this variance is a function of the assigned weight and the guaranteed lag bounds, as we later show. When a unit weight is assigned, a virtual processor perfectly imitates a dedicated processor.

Related concepts. Using server tasks (*i.e.*, virtual processors) to multiplex several applications onto a single platform is a relatively old idea. For instance, in 1989, Tucker and Gupta suggested using virtual processors to more seamlessly support workload changes when multiplexing parallel applications onto a multiprocessor Tucker and Gupta (1989). In addition, thread packages, which are now commonly available, are a direct application of the virtual-processor concept. For instance, Java programs are executed on a virtual processor referred to as the *Java virtual machine* (or JVM) Bollella (2000).

Server tasks are a central concept in work on *open systems*. In open systems, independently developed applications share one or more physical processors and must be isolated from each other. By far, the most-investigated real-time approach is the use of periodic and sporadic server tasks. Abeni and Buttazzo's constant-bandwidth server (CBS) approach (Abeni and Buttazzo, 1998) uses such server tasks to allocate processor time to collections of one-shot jobs and non-real time applications without compromising real-time guarantees. In recent years, this work has even been extended to support resource sharing across servers (Caccamo and Sha, 2001; Lamastra et al., 2001).

In (Shin and Lee, 2003), Shin and Lee proposed a similar approach that uses periodic server tasks to schedule periodic workloads. The benefit of using the same task model for server and client tasks is that the system can be arranged into a task hierarchy of arbitrary depth, *i.e.*, a server task can be the client of another server task. Unfortunately, some loss inevitably results from the use of server tasks, as shown in (Shin and Lee, 2003). Since it is unclear how using a multi-level hierarchy will improve upon using an equivalent two-level hierarchy, it remains to be shown whether such composability is of any practical interest.

Unfortunately, effective use of CBS and similar approaches on a non-partitioned multiprocessor requires the ability to globally schedule periodic and sporadic tasks effectively. (Partitioning was mentioned earlier in Section 1.) Ironically, it was the inability to produce an effective means for accomplishing this same goal that prompted research into the Pfair approach. In addition, to the best of our knowledge, no analysis allows for the use of CBS and similar approaches when tasks executing on remote processors can potentially interfere with the execution of the servers. Hence, the scope of these approaches is limited even on partitioned multiprocessors. Since Pfair scheduling is not based on the periodic and sporadic task models, these approaches are not a viable solution to the problem considered in this paper.

On the other hand, prior work on the resource partition model (Lipari and Bini, 2003; Feng and Mok, 2002; Mok and Feng, 2001; Mok et al., 2001) is applicable here. Under this model, a server task is characterized by two parameters: its guaranteed bandwidth and its maximum execution delay. The advantage of such a server model is that it divides the problem of group-based scheduling into two independent sub-problems. First, the system scheduler must guarantee that the server is scheduled in a manner that respects its parameters. Second, the server's internal scheduler must schedule its clients so that all client constraints are respected whenever the server is scheduled correctly.

Unfortunately, this additional layer of abstraction also introduces the primary limitation of this approach. Specifically, analysis under this model does not consider the approach used to schedule the server tasks. Instead, the amount of processor time allocated to server tasks is estimated based only on the model parameters assigned to the server (*e.g.*, guaranteed bandwidth and maximum delay) and on the assumption that these parameters are always respected. Basing these estimates on the server model instead of the actual scheduling parameters of the servers (*e.g.*, the task weights) and the scheduling approach in use (*e.g.*, Pfair scheduling) almost certainly results in consistent underestimation, and hence in more loss. Due to this limitation, we do not consider the use of this model here.

Supertasking and partitioning. Supertasking extends the one-to-one relationship between virtual processors and tasks (considered in (Baruah et al., 1996)) into a one-to-many relationship. In doing so, a new problem is introduced: how should tasks be grouped? In practice, tasks may not always be implicitly grouped. In such cases, tasks can be artificially grouped in order to reduce contention and overhead. Notice that dividing tasks among groups bears a strong resemblance to partitioning approaches.

The primary difference between supertasking and partitioning is that partitioning binds tasks to physical processors, while supertasking binds tasks to virtual processors. Under partitioning, exactly M processors are available, each with a fixed capacity.[3] Neither of these values (*i.e.*, the processor count and capacity) can be varied. On the other hand, the number of supertasks is unrestricted and capacities (*i.e.*, weights) can be assigned *after* making task assignments. Hence, the assignment of tasks to supertasks can be accomplished through an algorithm of the following form:

1. Assume initial parameter values (*e.g.*, weights, blocking estimates, *etc.*) for all tasks.
2. Create one empty supertask per task.
3. Assign $\mathcal{S}.w = 1$ to each supertask $\mathcal{S}$.
4. Apply a heuristic to assign each task to some supertask.

[3]The capacity is the maximum schedulable utilization, which is determined by the scheduling algorithm and task models.

5. Remove all empty (unused) supertasks.
6. Update parameter values of all tasks based on assignments.
7. Update supertask weights based on component-task parameters.
8. If any supertask weight exceeds unity, start over at Step 2.

Step 7 in the above algorithm is the primary difference between supertasking and partitioning. In this step, supertask capacities are reduced to better match the requirements of the assigned component tasks. Such an action is clearly not possible when assigning tasks directly to physical processors.

3.1 The Cost of Supertasking

In order to quantify the cost of using supertasks, it is necessary to first define an ideal form to use as a baseline for comparison. In this paper, we consider the use of both quantum-based and fully preemptive scheduling within supertasks. We define an ideal form of supertasking for each of these alternatives below.

Quantum-based supertasking. Under quantum-based supertasking, we assume that all component tasks are Pfair tasks. As observed by Moir and Ramamurthy (Moir and Ramamurthy, 1999), a quantum-based supertask would ideally be granted a weight equal to the cumulative weight of its component tasks. Letting $\mathcal{S}.I_Q$ denote this ideal weight results in the definition given below.

$$\mathcal{S}.I_Q \stackrel{\text{def}}{=} \sum_{T\in\mathcal{S}} T.w \tag{8}$$

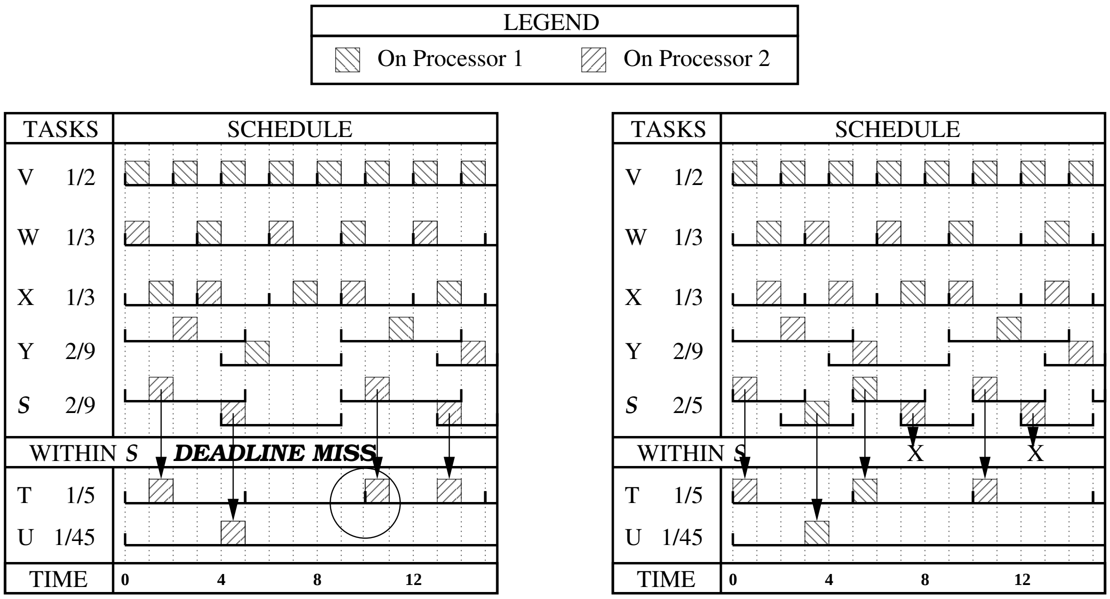
Hence, the overhead resulting from the use of a quantum-based supertask is given by $\mathcal{S}.w - \mathcal{S}.I_Q$. We refer to this overhead as *inflation* or *reweighting* overhead.

Fully preemptive supertasking. Under fully preemptive supertasking, we assume that all tasks are either periodic or sporadic tasks. In this case, we must first consider the relationship between weight and utilization, due to the fact that component tasks are not characterized by weights. Over an interval of length L, a task with weight w is allocated approximately $w \cdot L$ quanta, which results in a total allocation of $(w \cdot L) \cdot Q$. Hence, the utilization of the task over the interval is given by $\frac{(w\cdot L)\cdot Q}{L}$, which simplifies to $w \cdot Q$. It follows that a utilization of u is achieved by a weight of $\frac{u}{Q}$.

We now use this relationship to define the ideal weight. Ideally, a supertask would be assigned the smallest weight necessary to ensure the total utilization of its component tasks. This utilization is given by

$$\mathcal{S}.u \stackrel{\text{def}}{=} \sum_{T\in\mathcal{S}} T.u.$$

Using the relationship between weight and utilization, the ideal weight, denoted $\mathcal{S}.I_P$, is defined as shown



(a) **(b)**

Figure 3: Two-processor schedule with a **(a)** normal and **(b)** reweighted supertask $\mathcal{S}$.

below.

$$\mathcal{S}.I_P \stackrel{\text{def}}{=} \frac{\mathcal{S}.u}{Q} \tag{9}$$

Hence, the overhead produced by a fully preemptive supertask is given by $\mathcal{S}.w - \mathcal{S}.I_P$.

3.2 Understanding Failures

To see why supertasking (as proposed by Moir and Ramamurthy (Moir and Ramamurthy, 1999)) can fail, consider the two-processor Pfair schedule shown in Figure 3. The task set consists of four Pfair tasks ($\mathbf{V} = \mathbf{PF}(\frac{1}{2})$, $\mathbf{W} = \mathbf{PF}(\frac{1}{3})$, $\mathbf{X} = \mathbf{PF}(\frac{1}{3})$, and $\mathbf{Y} = \mathbf{PF}(\frac{2}{9})$) and one supertask $\mathcal{S}$ that represents the two component tasks $\mathbf{T} = \mathbf{PF}(\frac{1}{5})$ and $\mathbf{U} = \mathbf{PF}(\frac{1}{45})$ (shown in the lower region). Pfair global scheduling is assumed. In Figure 3(a), $\mathcal{S}$ competes with its ideal weight, *i.e.*, $\mathcal{S}.w = \mathcal{S}.I_Q = \frac{1}{5} + \frac{1}{45} = \frac{2}{9}$. All scheduling decisions in the upper (respectively, lower) region are consistent with the PD^2 (respectively, EPDF) policy. (Under EPDF, subtasks with earlier pseudo-deadlines are given higher priority.)

As the schedule shows, $\mathbf{T}$ misses a pseudo-deadline at time 10. This is because no quantum is allocated to $\mathcal{S}$ in the interval $[5, 10)$. In general, component tasks may violate their timing constraints whenever there exists an interval $[t_1, t_2)$ over which the total processor time required by the component tasks, denoted *demand*$(\mathcal{S}, t_1, t_2)$, exceeds the minimum amount of processor time guaranteed to the supertask, denoted *supply*$(\mathcal{S}, t_1, t_2)$. Observe that $[5, 10)$ is such an interval since *demand*$(\mathcal{S}, 5, 10) = Q$ due to $\mathbf{T}_2$, while *supply*$(\mathcal{S}, 5, 10) = 0$. To ensure the timeliness of component tasks, it is sufficient (though not necessary in most cases) to guarantee that *supply*$(\mathcal{S}, t_1, t_2) \geq$ *demand*$(\mathcal{S}, t_1, t_2)$ over all intervals in which a violation could potentially occur. Selecting a supertask weight that provides such a guarantee, called *reweighting*, is the focus of this paper.

Figure 3(b) illustrates how reweighting can ensure timeliness. In this schedule, $\mathcal{S}.w$ has been increased to $\frac{2}{5}$, resulting in an inflation of $\frac{2}{5} - \frac{2}{9} = \frac{8}{45}$. As shown, no component task violates its timing constraints. However, an unfortunate side effect of reweighting is that a supertask will inevitably be allocated more processor time than its component tasks can utilize; quanta marked with an "X" are allocated to the supertask but cannot be used.

3.3 Example Scenarios

One advantage of the reweighting methodology presented here is the ease with which it can be adapted to new scheduling scenarios. Unfortunately, since each scenario typically requires unique reasoning, it is not possible to derive formulas that can be universally applied in all scenarios. To facilitate the presentation, we have tried to use variable and function definitions to isolate scenario-specific details from the parts of the methodology that are common to all scenarios.

However, explaining the methodology without providing any concrete examples to illustrate each step can also be confusing. As a compromise, we have chosen two example scenarios that will be used to illustrate the application of our methodology (*i.e.*, we demonstrate the derivation of the scenario-specific details for each of these scenarios). These examples have been selected to highlight common problems and to provide a reasonable coverage of the issues involved when reweighting. (Additional examples can be found in (Holman, 2004).) Specifically, we consider the following scenarios:

Scenario 1: *Quantum-based EPDF scheduling* (QB-EPDF)

Component tasks are Pfair tasks, and are scheduled by a quantum-based supertask. Subtasks are prioritized by the EPDF policy. The global scheduler is assumed to respect Pfairness, *i.e.*, $\beta_- = \beta_+ = 1$ and $\epsilon_r = \epsilon_d = 0$.

Scenario 2: *EDF scheduling with nonpreemptable code segments* (FP-EDF-NP)

Component tasks are periodic and sporadic tasks that never suspend, but may execute non-preemptably (with respect to other component tasks).[4] Tasks are scheduled by a fully preemptive EDF policy within the supertask, *i.e.*, tasks with earlier job deadlines are given higher priority. For this scenario, we let $U.v$ denote an upper bound on the execution requirement of any nonpreemptable code segment of task U. This scenario is a generalization of fully nonpreemptive scheduling (within the supertask), which can be achieved by letting $U.v = U.e$ for all $U \in \mathcal{S}$. Similarly, fully preemptive scheduling without non-preemptable code segments can be achieved by letting $U.v = 0$ for all $U \in \mathcal{S}$.

[4] We do *not* consider making the supertask nonpreemptable since that would also impact global scheduling.

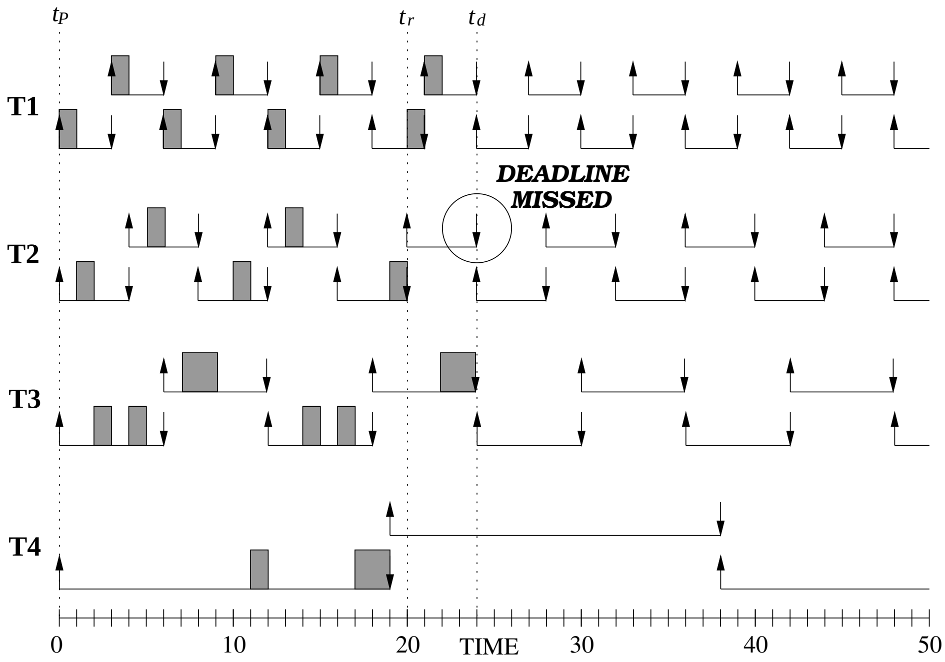


Figure 4: Sample DBA schedule consisting of four synchronous periodic tasks: $\mathbf{T1} = \mathbf{P}(0, 1, 3, 3)$, $\mathbf{T2} = \mathbf{P}(0, 1, 4, 4)$, $\mathbf{T3} = \mathbf{P}(0, 2, 6, 6)$, and $\mathbf{T4} = \mathbf{P}(0, 3, 19, 19)$. The task set is scheduled using a fully preemptive EDF policy and experiences its first violation at time 24, as shown.

4 Deriving a Schedulability Condition

We next derive a sufficient schedulability condition for a set of component tasks. To accomplish this, we present a framework for analysis based on uniprocessor *demand-based* analysis (DBA). When discussing DBA, we use the term *request* to refer to any request for processor time issued by a task (*i.e.*, either a job or a subtask depending on the scenario).

DBA on a dedicated uniprocessor. We begin by illustrating the ideas underlying DBA with an example. Figure 4 shows a schedule in which four independent, synchronous (*i.e.*, $T.\phi = 0$) periodic tasks are scheduled on a dedicated uniprocessor using a fully preemptive EDF policy. As shown, a timing violation (*i.e.*, deadline miss) occurs at time 24.

The goal of DBA is to characterize the state of the system leading up to a violation. Let R denote the request (job) that experiences the violation. (We will refer to R repeatedly throughout the section.) R is the sixth job of **T2** in Figure 4. Let t_d and t_r denote the times at which the violation occurs and at which R was released, respectively. In Figure 4, $t_d = R.d = 24$ and $t_r = R.a = 20$.

Now, consider the state of the system over $[t_r, t_d)$. First, due to the existence of R and the EDF-based prioritization of tasks, no idling occurred within $[t_r, t_d)$ and every job that executed in the interval had a deadline at or before t_d. Hence, Property BP, shown below, held throughout $[t_r, t_d)$.

Busy Period (BP): A ready or executing job exists that has priority at least that of R.

Unfortunately, characterizing the state of the system at t_r is difficult. Under DBA, this problem is addressed by simply moving the start of the interval under observation to an earlier point at which the state is known. Specifically, let t_P denote the earliest time within the interval $[0, t_r]$ for which a given property P holds throughout $[t_P, t_r)$.[5] In the DBA example, Property BP is P. It is often necessary to modify P to account for events that may occur at runtime, such as task suspensions. Indeed, we necessarily consider a different choice of P below to account for the scheduling of the supertask. Since $t_P = t_r$ trivially satisfies the requirement when no other points do, some valid choice of t_P always exists. In Figure 4, $t_P = 0$. Based on the selection criteria for t_P, $[t_P, t_d)$ also satisfies the properties stated above for $[t_r, t_d)$. In addition, the fact that Property BP does not hold immediately prior to t_P implies that all jobs executing in $[t_P, t_d)$ are released at or after time t_P.

It follows from the above observations that all processor time in $[t_P, t_d)$ is consumed by jobs with releases at or after t_P and with deadlines at or before t_d. The total processor time required to complete all such jobs is called *demand* and is denoted $\mathit{demand}(\tau, t_P, t_d)$. The available processor time in $[t_P, t_d)$ is then called *supply* and is denoted $\mathit{supply}(\tau, t_P, t_d)$. On a dedicated uniprocessor, supply is simply determined by the interval length, *i.e.*, $\mathit{supply}(\tau, t_P, t_d) = t_d - t_P$. The fact that a deadline miss occurs implies the following relationship (and a necessary condition for a deadline miss).

$$\mathit{demand}(\tau, t_P, t_d) > \mathit{supply}(\tau, t_P, t_d)$$

In Figure 4, $\mathit{demand}(\tau, t_P, t_d) = 25$ and $\mathit{supply}(\tau, t_P, t_d) = 24$.

Formalizing the condition. The goal of DBA is to derive a necessary condition for a timing violation to occur, *i.e.*, to find a condition C that satisfies

$$\textit{a violation occurs} \Rightarrow C.$$

Restating the relationship between supply and demand more formally produces the following definition for C:

$$C \stackrel{\text{def}}{=} (\exists t_P, t_d : t_d - t_P \geq \min\{\ T.p \mid T \in \tau\ \} : \mathit{demand}(\tau, t_P, t_d) > \mathit{supply}(\tau, t_P, t_d)).$$

The constraint $t_d - t_P \geq \min\{\ T.p \mid T \in \tau\ \}$ follows from the fact that $[t_P, t_d)$, defined above, always contains $[t_r, t_d)$. (The inequality $t_d - t_r \geq \min\{\ T.p \mid T \in \tau\ \}$ follows from the fact that $t_d - t_r = R.d$ and each task's relative deadline equals its period.) Taking the contrapositive of *a violation occurs* $\Rightarrow C$

[5]Typically, P is selected to be a property that holds over $[t_r, t_d)$. However, this approach does not necessarily produce the best results. Unfortunately, effective selection of P requires insights into the scenario under consideration and some degree of intuition, *i.e.*, we are aware of no systematic method for selecting P.

produces the sufficient schedulability condition

$$\neg C \Rightarrow \textit{no violation occurs}.$$

Hence, no deadline misses can occur if the following condition is satisfied:

$$(\forall t_P, t_d : t_d - t_P \geq \min\{\ T.p \mid T \in \tau\ \} : \textit{demand}(\tau, t_P, t_d) \leq \textit{supply}(\tau, t_P, t_d)). \tag{10}$$

Generalizing DBA for supertasking. Applying DBA to supertasks follows the same steps given above. However, several modifications are necessary. First, Property BP cannot be applied effectively because it does not account for the scheduling of the supertask. For the example scenarios considered here, we consider Property SBP, shown below.

Supertask Busy Period (SBP): Whenever $\mathcal{S}$ is scheduled, an eligible or executing request of a component task exists that has priority at least that of R.

Notice that the only significant difference between Property SBP and Property BP is that attention is restricted to times at which $\mathcal{S}$ is executing. To the best of our knowledge, all properties proposed for DBA on a dedicated uniprocessor can be similarly adapted to supertasking with relative ease. Second, since component tasks can only execute when $\mathcal{S}$ is scheduled, it follows that $\textit{supply}(\mathcal{S}, t_P, t_d) = \textit{received}(\mathcal{S}, t_P, t_d)$.

Bounding demand. The most difficult part of DBA is deriving bounds on the demand generated by the tasks. In the DBA example given in Figure 4, all demand was *mandatory* in the sense that the jobs contributing to the demand existed only within the interval $[t_P, t_d)$. Hence, successful scheduling required that all such jobs, called *mandatory* jobs, be executed within $[t_P, t_d)$. For instance, no combination of scheduling policy and synchronization protocols can successfully schedule the task set without executing the third job **T3** within $[0, 24)$.

Real task systems are typically more complex, which results in additional demand being introduced into each interval. We refer to such demand as *circumstantial demand*. Circumstantial demand is generated by dependencies between requests (*e.g.*, precedence constraints, resource sharing, *etc.*) and the use of suboptimal policies (*e.g.*, using rate-monotomic (RM) scheduling[6] on a uniprocessor, allowing non-preemptable execution, *etc.*).

The circumstantial demand stemming from the latter source is the result of how policy choices can impact the difficulty of scheduling. For instance, if a fully preemptive RM policy is used on a dedicated uniprocessor, then a job J with a later deadline than R may be prioritized over R. Hence, the choice of scheduling policy imposes the unnecessary constraint that J be executed before R (if both requests are pending). When the goal of scheduling is to meet deadlines, such a constraint is illogical. Indeed, this policy choice actually

[6]Under RM scheduling, tasks with smaller periods are given higher priority.

makes scheduling more difficult due to the fact that more processor time is needed to guarantee that R's deadline is met (as compared to that required when an optimal prioritization is used). Since it is not necessary to schedule J in $[t_P, t_d)$, the demand contributed by J is circumstantial demand. Similarly, a job J can contribute to the circumstantial demand by holding a lock that is needed by a mandatory job or by initiating a non-preemptive code segment immediately prior to t_P.

To distinguish between these two forms of demand, we decompose $\mathit{demand}(\mathcal{S}, t_P, t_d)$ into mandatory demand, denoted $\mathit{demand}_M(\mathcal{S}, t_P, t_d)$, and circumstantial demand, denoted $\mathit{demand}_C(\mathcal{S}, t_P, t_d)$. Hence, $\mathit{demand}(\mathcal{S}, t_P, t_d) = \mathit{demand}_M(\mathcal{S}, t_P, t_d) + \mathit{demand}_C(\mathcal{S}, t_P, t_d)$. Since the QB-EPDF scenario schedules subtasks in deadline order and considers only independent tasks, $\mathit{demand}_C(\mathcal{S}, t_P, t_d) = 0$ under this scenario. However, circumstantial demand does arise under the FP-EDF-NP scenario due to non-preemptable execution.

Schedulability condition. We now state the schedulability condition that will be used to drive the reweighting process, which is discussed later in Section 5. The condition, which generalizes (10), is shown below.

$$(\forall t, L : L \geq L_0 : \mathit{demand}_M(\mathcal{S}, t, t+L) + \mathit{demand}_C(\mathcal{S}, t, t+L) \leq \mathit{supply}(\mathcal{S}, t, t+L)). \tag{11}$$

(11) is derived from (10) by replacing t_P and t_d with t and $t + L$, respectively. (The purpose of this substitution is to make the interval length, L, an explicit parameter.) L_0 is used to denote the shortest interval over which a request can be released and experience a violation. As explained above, $[t, t + L)$ must always contain these two points. All terms in (11) are scenario-specific. In the subsections that follow, we derive bounds for these terms for each of the example scenarios.

4.1 Defining L_0

Deriving L_0 is straightforward as it depends only on the shortest interval that can contain a release and the associated violation.

Scenario 1 (QB-EPDF). This scenario focuses on subtasks with Pfair windows. Hence, by the $n = 1$ case of Lemma 1, L_0 can be defined as shown below.

$$L_0 = \min\left\{ \left\lceil \frac{1}{U.w} \right\rceil \;\middle|\; U \in \mathcal{S} \right\} \tag{12}$$

Scenario 2 (FP-EDF-NP). Under this scenario, each request is a job. Since periods are assumed to equal deadlines, L_0 is defined as shown below.

$$L_0 = \min\{\ U.p \mid U \in \mathcal{S}\ \}. \tag{13}$$

4.2 Defining *supply*$(\mathcal{S}, t, t + L)$

We now derive a bound on the minimum amount of processor time guaranteed to a task over any interval of length L. Since a supertask behaves like any other Pfair task, *supply*$(\mathcal{S}, t, t + L)$ can be defined using such a bound. The following theorems bound the amount of processor time guaranteed over intervals $[0, t)$ and $[t, t + L)$, respectively, that align to slot boundaries at both ends. These theorems are stated without proof. (The proofs can be found in an appendix.)

Theorem 2 *The amount of processor time received by a task T over the interval $[0, t)$, where t is an integer, under scheduling characterized by β_-, β_+, ϵ_r, and ϵ_d, is bounded as shown below.*

$$(\lfloor T.w \cdot (t - \epsilon_d) - \beta_+ \rfloor + 1) \cdot Q \leq \mathit{received}(T, 0, t) \leq (\lceil T.w \cdot (t + \epsilon_r) + \beta_- \rceil - 1) \cdot Q$$

Theorem 3 *The amount of processor time received by a task T over the interval $[t, t + L)$, where t and L are integers, under scheduling characterized by β_-, β_+, ϵ_r, and ϵ_d, is bounded as shown below.*

$$(\lfloor T.w \cdot (L - \epsilon) - \beta \rfloor + 1) \cdot Q \leq \mathit{received}(T, t, t + L) \leq (\lceil T.w \cdot (L + \epsilon) + \beta \rceil - 1) \cdot Q$$

Using the above results, we now bound *supply*$(T, t, t + L)$ in the pair of corollaries given below. Each corollary corresponds to a case that can arise under supertasking. Specifically, the interval under inspection can align to slot boundaries on both ends (Corollary 1), on only one end (not considered here), or on neither end (Corollary 2). (See (Holman, 2004) for a discussion of the omitted case.) Due to quantum-based scheduling, Corollary 1 applies to the QB-EPDF scenario. However, since job releases and deadlines may lie off slot boundaries, Corollary 2 must be used for the FP-EDF-NP scenario.

Corollary 1 *The supply of a supertask $\mathcal{S}$ that does not delay any subtask releases over the interval $[t, t + L)$, where t and L are integers, while executing under scheduling characterized by β_-, β_+, ϵ_r, and ϵ_d, satisfies*

$$\mathit{supply}(\mathcal{S}, t, t + L) \geq (\lfloor \mathcal{S}.w \cdot (L - \epsilon) - \beta \rfloor + 1) \cdot Q.$$

Proof. Since *supply*$(\mathcal{S}, t, t + L) =$ *received*$(\mathcal{S}, t, t + L)$, the corollary follows trivially from Theorem 3. □

Corollary 2 *The supply of a supertask $\mathcal{S}$ that does not delay any subtask releases over any interval $[t, t + L)$, while executing under scheduling characterized by β_-, β_+, ϵ_r, and ϵ_d, satisfies*

$$\mathit{supply}(\mathcal{S}, t, t + L) \geq (\lfloor \mathcal{S}.w \cdot (\lfloor L \rfloor - 1 - \epsilon) - \beta \rfloor + 1) \cdot Q.$$

Proof. The subinterval $[\lceil t \rceil, \lfloor t + L \rfloor)$ always has length at least $\lfloor L \rfloor - 1$ and aligns to slot boundaries on both ends. Hence, the corollary follows from Theorem 3. □

Corollary 2 reflects a straightforward estimate of the supertask supply. Specifically, when either of t or $t + L$ lies off of a slot boundary, this estimate pessimistically assumes that the supertask is not scheduled within those partially overlapped slots. This assumption is implied by the proof, which only considers processor time allocated in the subinterval $[\lceil t \rceil, \lfloor t + L \rfloor)$. We consider this estimate because it makes for a simpler example. (Recall that these scenarios are presented only to provide examples of using our framework.) This pessimism can be avoided through the use of a more complex analysis that also takes into account the amount of processor time allocated over the subintervals $[\lceil t \rceil, \lceil t + L \rceil)$ and $[\lfloor t \rfloor, \lfloor t + L \rfloor)$.

4.3 Defining $demand_M(\mathcal{S}, t, t + L)$

In this subsection, we derive bounds on the mandatory demand of a component task set. Indeed, the bounds that we derive apply to any task set. For this reason, the results derived below do not refer specifically to component tasks, but rather to an arbitrary set of tasks.

One advantage to separating demand into its mandatory and circumstantial components is that bounds on mandatory demand depend only on the timing constraint (*e.g.*, job deadline, subtask deadlines, *etc.*) used. Hence, bounds are universal across scheduling policies. Our example scenarios require consideration of two cases: subtask demand (QB-EPDF) and job demand (FP-EDF-NP). These cases are addressed by Corollaries 3 and 4, respectively, which are given below.

Scenario 1 (QB-EPDF). Subtask demand is bounded by the following theorem and corollary.

Theorem 4 *When scheduling subtasks of tasks described by the Pfair task model, the mandatory demand generated by a task T, over the interval $[t, t + L)$, where t and L are integers, is upper-bounded by*

$$demand_M(T, t, t + L) \leq \lfloor T.w \cdot L \rfloor \cdot Q.$$

Proof. The mandatory demand of T over $[t, t + L)$ can be computed as the difference between T's minimum allocation at $t + L$ less its maximum allocation at t. Applying Theorem 2 with $\beta_+ = \beta_- = 1$ and $\epsilon_r = \epsilon_d = 0$ produces the following formula:

$$demand_M(T, t, t + L) = (\lfloor T.w \cdot (t + L) \rfloor - \lceil T.w \cdot t \rceil) \cdot Q.$$

Rewriting the first term yields the following formula.

$$demand_M(T, t, t + L) = (\lfloor T.w \cdot t + T.w \cdot L \rfloor - \lceil T.w \cdot t \rceil) \cdot Q$$

Since $\lfloor a + b \rfloor \leq \lceil a \rceil + \lfloor b \rfloor$, it follows that

$$demand_M(T, t, t + L) \leq (\lfloor T.w \cdot L \rfloor + \lceil T.w \cdot t \rceil - \lceil T.w \cdot t \rceil) \cdot Q.$$

Simplifying this bound establishes the theorem. □

Corollary 3 *When scheduling subtasks of tasks described by the Pfair task model and its variants, the following formula gives a series of progressively looser upper bounds on the total mandatory demand generated by the component task set of the supertask $\mathcal{S}$ over the interval $[t, t+L)$, where t and L are integers, and $\mathcal{S}_L = \{\ T \mid T \in \mathcal{S} \wedge T.c \leq L\ \}$.*

$$\begin{aligned}
demand_M(\mathcal{S}, t, t+L) &= \sum_{T \in \mathcal{S}} demand_M(T, t, t+L) \\
&\leq Q \sum_{T \in \mathcal{S}_L} \lfloor T.w \cdot L \rfloor \\
&\leq Q \left\lfloor \sum_{T \in \mathcal{S}_L} (T.w \cdot L) \right\rfloor \\
&\leq Q \sum_{T \in \mathcal{S}_L} (T.w \cdot L)
\end{aligned}$$

Proof. The first inequality follows trivially from Theorem 4. The second and third inequalities follow from the properties $\lfloor a \rfloor + \lfloor b \rfloor \leq \lfloor a + b \rfloor$ and $\lfloor a \rfloor \leq a$, respectively. □

Scenario 2 (FP-EDF-NP). Finally, the next theorem and corollary bound the mandatory demand produced by jobs.

Theorem 5 *When scheduling jobs of a periodic or sporadic task, the mandatory demand generated by a task T over the interval $[t, t+L)$ is upper-bounded by*

$$demand_M(T, t, t+L) \leq \left\lfloor \frac{L}{T.p} \right\rfloor \cdot T.e$$

when T satisfies $T.c \leq L$. When $T.c > L$, T generates no mandatory demand.

Proof. At most $\left\lfloor \frac{L}{T.p} \right\rfloor$ jobs are enclosed in an interval of length L. The bound given in the theorem follows trivially from the fact that each job requires at most $T.e$ units of processor time. □

Corollary 4 *When scheduling jobs of a periodic or sporadic task, the following formula gives a series of progressively looser upper bounds on the total mandatory demand generated by the component task set of the supertask $\mathcal{S}$ over the interval $[t, t+L)$.*

$$\begin{aligned}
demand_M(\mathcal{S}, t, t+L) &= \sum_{T \in \mathcal{S}} demand_M(T, t, t+L) \\
&\leq \sum_{T \in \mathcal{S}} \left(\left\lfloor \frac{L}{T.p} \right\rfloor \cdot T.e \right) \\
&\leq \sum_{T \in \mathcal{S}} (T.u \cdot L)
\end{aligned}$$

Proof. The first inequality follows from Theorem 5. The second follows from $\lfloor a \rfloor \leq a$. □

4.4 Defining $demand_C(\mathcal{S}, t, t + L)$

Unfortunately, bounding circumstantial demand requires unique reasoning for each scenario. (Indeed, this is why we use the term "circumstantial.") We now bound this demand for each of the example scenarios. All analysis presented in this section assumes the use of Property SBP, given earlier. As before, let $[t, t + L)$ denote the interval under consideration.

Scenario 1 (QB-EPDF). This step is trivial due to the fact that all tasks are independent and never suspend. Because requests are processed in deadline order, the mandatory demand is always serviced before all other requests. Hence, no circumstantial demand exists.

Scenario 2 (FP-EDF-NP). Under the FP-EDF-NP scenario, a job of some task U with priority lower than the job experiencing the violation (*i.e.*, R) can begin nonpreemptable execution immediately before t and, consequently, avoid preemption at t. By Property SBP, such a job, if one indeed exists, does not need to complete until after $t + L$. Since this job must have been released and executed prior to t to initiate its nonpreemptable execution, it follows that $U.p > L$ holds. Furthermore, at most one such job can execute within $[t, t + L)$ since two component tasks cannot be executing nonpreemptably at the same time. Hence, circumstantial demand under Scenario 2 is upper bounded as shown below.

$$demand_C(\mathcal{S}, t, t + L) \leq \max\{\ U.v \mid U \in \mathcal{S} \wedge U.p > L\ \} \tag{14}$$

Observe that $demand_C(\mathcal{S}, t, t + L) = 0$ for all $L \geq \max\{\ U.p \mid U \in \mathcal{S}\ \}$. Indeed, circumstantial-demand terms often go to zero as L increases. We refer to such terms as *transient* demand.

4.5 Summary

The table shown below summarizes which of the results presented in the previous subsections define the component parts of (11) under each scenario.

Scenario	L_0	*supply*	$demand_M$	$demand_C$
1	(12)	Corollary 1	Corollary 3	*N/A*
2	(13)	Corollary 2	Corollary 4	(14)

5 Deriving a Reweighting Condition

In this section, we use the schedulability condition derived in the Section 4 to derive a reweighting condition. A reweighting condition consists of a set of *weight restrictions* (*i.e.*, lower bounds on the supertask's weight) such that satisfying all restrictions ensures the timeliness of component tasks. A reweighting condition takes the following abstract form:

$$\mathcal{S}.w \geq \max\{\ \Delta(\mathcal{S}, L) \mid L \in \mathcal{L}\ \}. \tag{15}$$

Condition (15) consists of two elements: the *reweighting function* $\Delta(\mathcal{S}, L)$ and the *testing set* $\mathcal{L}$. $\Delta(\mathcal{S}, L)$ is the minimum weight needed to ensure that no timing violation occurs over any interval of length L, *i.e.*, $\Delta(\mathcal{S}, L)$ is the weight restriction imposed by all intervals of length L. Let $\mathcal{S}.w_{\text{opt}}$ denote the smallest weight satisfying (15), as shown below.

$$\mathcal{S}.w_{\text{opt}} \stackrel{\text{def}}{=} \max\{\ \Delta(\mathcal{S}, L) \mid L \in \mathcal{L}\ \} \tag{16}$$

As in the previous section, we consider the derivation of each element in a separate subsection. Specifically, the three subsections of this section address the following issues:

1. deriving $\Delta(\mathcal{S}, L)$;

2. defining $\mathcal{L}$;

3. efficient generation of L values from $\mathcal{L}$.

5.1 Deriving $\Delta(\mathcal{S}, L)$

We begin by deriving the reweighting formula.

Scenario 1 (QB-EPDF). Filling in the terms of the inequality in (11) using Corollary 1 and the tightest bound provided by Corollary 3 yields the inequality shown below (after canceling the common Q term on both sides).

$$\sum_{T \in \mathcal{S}_L} \lfloor T.w \cdot L \rfloor \leq \lfloor \mathcal{S}.w \cdot (L - \epsilon) - \beta \rfloor + 1$$

By the property $\lfloor x \rfloor \leq \lfloor y \rfloor \Leftrightarrow \lfloor x \rfloor \leq y$, the above inequality is equivalent to

$$\sum_{T \in \mathcal{S}_L} \lfloor T.w \cdot L \rfloor \leq \mathcal{S}.w \cdot (L - \epsilon) - \beta + 1.$$

Rearranging to isolate $\mathcal{S}.w$ yields the following weight restriction (and definition of $\Delta(\mathcal{S}, L)$).

$$\mathcal{S}.w \geq \frac{\sum_{T \in \mathcal{S}_L} \lfloor T.w \cdot L \rfloor + \beta - 1}{L - \epsilon} \stackrel{\text{def}}{=} \Delta(\mathcal{S}, L)\,, \tag{17}$$

where L_0 is required to satisfy

$$L_0 > \epsilon. \tag{18}$$

Constraint (18) ensures that $L - \epsilon > 0$ for all $L \geq L_0$. When this constraint does not hold, it is not possible to reweight the supertask using the technique presented here. By (12), the constraint given in (18) can also be stated as shown below.

$$\min\left\{ \left\lceil \frac{1}{U.w} \right\rceil \;\middle|\; U \in \mathcal{S} \right\} > \epsilon \tag{19}$$

Since periods are expected to be much larger than the slot size and ϵ is expected to be small, (19) should seldom, if ever, not hold.

Scenario 2 (FP-EDF-NP). Again, filling in the terms of (11) using Corollary 2, (14), and the tightest bound provided by Corollary 4 yields the inequality shown below.

$$\sum_{T \in \mathcal{S}_L} \left(\left\lfloor \frac{L}{T.p} \right\rfloor \cdot T.e \right) + \upsilon_L \leq (\lfloor \mathcal{S}.w \cdot (\lfloor L \rfloor - 1 - \epsilon) - \beta \rfloor + 1) \cdot Q$$

where

$$\upsilon_L \stackrel{\text{def}}{=} \max\{\; U.\upsilon \mid U \in \mathcal{S} \wedge U.p > L \;\}\,. \tag{20}$$

Rewriting this inequality yields the equivalent form shown below.

$$\sum_{T \in \mathcal{S}_L} \left(\left\lfloor \frac{L}{T.p} \right\rfloor \cdot \frac{T.e}{Q} \right) + \frac{\upsilon_L}{Q} - 1 \leq \lfloor \mathcal{S}.w \cdot (\lfloor L \rfloor - 1 - \epsilon) - \beta \rfloor$$

This condition is satisfied if

$$\left\lceil \sum_{T \in \mathcal{S}_L} \left(\left\lfloor \frac{L}{T.p} \right\rfloor \cdot \frac{T.e}{Q} \right) + \frac{\upsilon_L}{Q} \right\rceil - 1 \leq \mathcal{S}.w \cdot (\lfloor L \rfloor - 1 - \epsilon) - \beta$$

is satisfied. Notice that this latter condition is slightly stronger. Rewriting this inequality to isolate $\mathcal{S}.w$ produces the following weight restriction (and definition of $\Delta(\mathcal{S}, L)$):

$$\mathcal{S}.w \geq \frac{\left\lceil \sum_{T \in \mathcal{S}_L} \left(\left\lfloor \frac{L}{T.p} \right\rfloor \cdot \frac{T.e}{Q} \right) + \frac{\upsilon_L}{Q} \right\rceil + \beta - 1}{\lfloor L \rfloor - 1 - \epsilon} \stackrel{\text{def}}{=} \Delta(\mathcal{S}, L)\,, \tag{21}$$

which requires

$$L_0 \geq \epsilon + 2. \tag{22}$$

Again, the above restriction ensures that $\lfloor L \rfloor - 1 - \epsilon > 0$ for all $L \geq L_0$. By (13), this constraint is equivalent to that given below.

$$\min\{\; U.p \mid U \in \mathcal{S} \;\} \geq \epsilon + 2. \tag{23}$$

As with (19), (23) should seldom, if ever, be violated.

The impact of the mandatory-demand bound. Both derivations given above are based on the tightest bounds on mandatory demand provided by Corollaries 3 and 4. Alternatively, one of the looser bounds provided by these corollaries could have been used. The benefit of using a looser bound is that it speeds the reweighting process. This is due to the fact that the running time of the reweighting algorithm (presented later in the paper) is proportional to the reweighting overhead. Since looser mandatory-demand bounds overestimate demand, they produce more inflation. Consequently, reweighting requires less time.

Unfortunately, using looser bounds is a poor method for reducing the execution time of the reweighting algorithm because the impact of the bound on both the execution time and the reweighting overhead is difficult to predict. For this reason, we do *not* recommend the use of looser bounds or consider their use beyond this point. As an alternative, the reweighting algorithm has been equipped with input parameters that allow manipulation of the speed-versus-accuracy trade-off in a controlled and predictable way.

5.2 Defining $\mathcal{L}$

As a starting point, consider letting $\mathcal{L} \stackrel{\text{def}}{=} \{ \; L \mid \; L \geq L_0 \wedge L \in \mathcal{N} \; \}$ when using quantum-based supertasking and $\mathcal{L} \stackrel{\text{def}}{=} \{ \; L \mid \; L \geq L_0 \; \}$ when using fully preemptive supertasking. (In the first set definition, $\mathcal{N}$ denotes the set of natural numbers.) These definitions ensure the correctness of the reweighting condition, but include an unnecessarily high number of points in $\mathcal{L}$.

Scenario 1 (QB-EPDF). Consider the reweighting function given in (17) and a range $L \in [L_1, L_2)$ over which the numerator remains constant. Due to the L parameter in the denominator, the function's value can only decrease as L takes on larger values within $[L_1, L_2)$. Hence, a search for the maximum value requires only that the smallest L value from the testing set in $[L_1, L_2)$ need be checked. In (17), changes are caused by the argument to the floor operation; the k^{th} increase of T's term in the summation occurs at the following value of L.

$$L(k) \stackrel{\text{def}}{=} \min \{ \; L \;\mid T.w \cdot L \geq k \; \} = \frac{k}{T.w}$$

However, $L(k)$ may not be in $\mathcal{N}$. Applying this additional restriction produces the set $\{ \; \lceil L(k) \rceil \mid \; k \geq 1 \; \}$, which can also be expressed as

$$\left\{ \; \left\lceil \frac{k}{T.w} \right\rceil \; \middle| \; k \geq 1 \; \right\}.$$

The definition of $\mathcal{L}$ for Scenario 1 is then the result of unioning these per-task testing sets, as shown below.

$$\mathcal{L} \stackrel{\text{def}}{=} \bigcup_{T \in \mathcal{S}} \left\{ \; \left\lceil \frac{k}{T.w} \right\rceil \; \middle| \; k \geq 1 \; \right\} \tag{24}$$

Scenario 2 (FP-EDF-NP). In Scenario 2, the process of defining $\mathcal{L}$ follows the same steps. Specifically, the numerator of the reweighting function given in (21) changes only due to changes in either a floor term of the summation or the v_L term.

First, consider the floor terms. As before, we compute the value of L corresponding to the k^{th} increase in the term's value, as shown below.

$$L(k) \stackrel{\text{def}}{=} \min\left\{ L \;\middle|\; \frac{L}{T.p} \geq k \right\} = k \cdot T.p.$$

Since L is not required to be an integer when using fully preemptive scheduling, the per-task sets $\{\ L(k) \mid k \geq 1\ \}$ are simply unioned to obtain the following definition of $\mathcal{L}$.

$$\mathcal{L} \stackrel{\text{def}}{=} \bigcup_{T \in \mathcal{S}} \{\ k \cdot T.p \mid k \geq 1\ \} \tag{25}$$

It remains to account for changes in the numerator caused by υ_L. Consider the definition of υ_L given in (20). It follows from the $U.p > L$ condition in the set definition that if υ_L differs from $\upsilon_{L-\varepsilon}$, for arbitrarily small ε, then $U.p = L$ for the task U that defines $\upsilon_{L-\varepsilon}$, *i.e.*, U is eliminated from consideration at L but is still being considered at $L - \epsilon$. Hence, the set of L values at which the υ_L term may change its value is given by $\{\ T.p \mid T \in \mathcal{S}\ \}$. Since this set is a subset of the previous definition of $\mathcal{L}$, it follows that the definition given in (25) is sufficient in this scenario.

5.3 Generating L Values from $\mathcal{L}$

In this subsection, we present an efficient algorithm for generating a monotonically increasing sequence of L values from $\mathcal{L}$ such that no values in $\mathcal{L}$ are skipped. This algorithm is used as a subroutine by our reweighting algorithm (presented later).

The generator pseudo-code is shown in Figure 5. The algorithm consists of two routines: `InitGenerator` and `Generate`. `InitGenerator` initializes the generator, while `Generate` generates the next L value in the sequence. A detailed description follows.

Data structures. The set of L values generated by a specific task T is represented by a *nodetype* record. The $f(n)$ function field defines the L-value generating function, *i.e.*, the function implied by either (24) or (25). For instance, under Scenario 2, $f(n) = n \cdot T.p + T.c$, as suggested by (25). The *value* field stores the next L value in T's sequence. The minimum of these candidate values is determined by storing the records in a min-ordered heap by *value*.

Detailed description. The generator is initialized by a call to `InitGenerator`. In line 1, the heap is created. In line 3–5, each task's record is created and fields are initialized; the record is then stored in the heap in line 6.

`Generate` retrieves the smallest unreported L value in $\mathcal{L}$. Line 7 identifies this value. Lines 8–12 update the heap entry of each task that has the selected L value. The selected value is returned in line 13. For both routines, $O(|\mathcal{S}| \log |\mathcal{S}|)$ time complexity can be achieved by using a binomial heap.

```
typedef nodetype
    record
        begin
            L: real;
            k: integer;
            f(n: integer): real function
        end

var
    H: min-ordered heap of nodetype
            ordered by L;
    node: nodetype

procedure InitGenerator(S)
    H := MakeHeap();
    for each T ∈ S do
        node := MakeNode(T);
        node.k := 1;
        node.L := node.f(node.k);
        Insert(H,node)
    od

procedure Generate() returns real
    L := Min(H).L;
    while Min(H).L = L do
        node := ExtractMin(H);
        node.k := node.k + 1;
        node.L := node.f(node.k);
        Insert(H,node)
    od;
    return L
```

Figure 5: Algorithm for generating L values in sequence based from a definition of $\mathcal{L}$.

6 Selecting a Safe Weight

The $\mathcal{L}$ definitions given above imply that reweighting may require an unbounded number of computations. In this section, we address this issue by presenting a technique for detecting when the process can safely terminate. The solution presented here not only detects termination, but also provides a means for *forcing* termination at the expense of a predictable amount of extra inflation.

Concept. Our approach is based on examining the behavior of a *bounding function* $\phi(\mathcal{S}, L)$ of $\Delta(\mathcal{S}, L)$. Specifically, $\phi(\mathcal{S}, L)$ must satisfy the following constraints:

Bounding Constraint (BC): $(\forall L : L \geq L_\phi : \phi(\mathcal{S}, L) \geq \Delta(\mathcal{S}, L))$.

Monotonicity Constraint (MC): $\phi(\mathcal{S}, L)$ is a monotonic function of L.

Property BC ensures that $\phi(\mathcal{S}, L)$ upper bounds $\Delta(\mathcal{S}, L)$ for all values of L at and after L_ϕ, which is called the *activation point*. Hence, if this property holds and $L \geq L_\phi$, then $\mathcal{S}.w \geq \phi(\mathcal{S}, L) \Rightarrow \mathcal{S}.w \geq \Delta(\mathcal{S}, L)$. As explained below, L_ϕ allows $\phi(\mathcal{S}, L)$ to more tightly bound $\Delta(\mathcal{S}, L)$ by skipping over smaller L values at which transient demand exists. We postpone a discussion of Property MC until after the derivation of $\phi(\mathcal{S}, L)$ is presented.

6.1 Deriving $\phi(\mathcal{S}, L)$

$\phi(\mathcal{S}, L)$ and L_ϕ can typically be defined using the following rules of thumb:

Rule 1: Let L_ϕ be the maximum of the following values:

- L_0 (*i.e.*, the smallest L value considered when reweighting);

- the smallest L value such that no transient demand exists in intervals of length $L' \geq L$.

The second value ensures that transient demand in $\Delta(\mathcal{S}, L)$ can be ignored when applying Rule 2.

Rule 2: Derive $\phi(\mathcal{S}, L)$ from $\Delta(\mathcal{S}, L)$ by replacing all non-continuous terms in $\Delta(\mathcal{S}, L)$ with continuous upper bounds. For instance, x and $x + 1$ are continuous upper bounds of $\lfloor x \rfloor$ and $\lceil x \rceil$, respectively.

Scenario 1 (QB-EPDF). In Scenario 1, no circumstantial demand exists. Hence, transient demand never exists and can be ignored. Applying Rule 1 yields

$$L_\phi \stackrel{\text{def}}{=} L_0. \tag{26}$$

Applying Rule 2 to (17) produces the following upper bound:

$$\frac{\sum_{T \in \mathcal{S}} T.w \cdot L + \beta - 1}{L - \epsilon}.$$

Reorganizing the terms produces the following equivalent form.

$$\sum_{T \in \mathcal{S}} T.w + \frac{\left(\sum_{T \in \mathcal{S}} T.w \right) \cdot \epsilon + \beta - 1}{L - \epsilon}$$

Applying (8) produces the following equivalent form:

$$\mathcal{S}.I_Q + \frac{\mathcal{S}.I_Q \cdot \epsilon + \beta - 1}{L - \epsilon}.$$

This leads to the following definition:

$$\phi(\mathcal{S}, L) \stackrel{\text{def}}{=} \mathcal{S}.I_Q + \frac{\Psi(\mathcal{S})}{L - \epsilon}, \tag{27}$$

where

$$\Psi(\mathcal{S}) \stackrel{\text{def}}{=} \mathcal{S}.I_Q \cdot \epsilon + \beta - 1. \tag{28}$$

By (18), the denominator of the second term in (27) is always positive. Hence, the behavior of $\phi(\mathcal{S}, L)$ as L $(\geq L_\phi)$ increases is determined by the value of $\Psi(\mathcal{S})$. We refer to $\Psi(\mathcal{S})$ as the *characteristic function* of $\phi(\mathcal{S}, L)$. Specifically, $\phi(\mathcal{S}, L)$ is decreasing when $\Psi(\mathcal{S}) > 0$, constant when $\Psi(\mathcal{S}) = 0$, and increasing when $\Psi(\mathcal{S}) < 0$. Since $\Psi(\mathcal{S})$ is independent of L, its value can be pre-computed prior to reweighting.

Scenario 2 (FP-EDF-NP). By Rule 1 and (20), L_ϕ is defined as shown below.

$$L_\phi \stackrel{\text{def}}{=} \max(L_0, \max\{\ T.p \mid T \in \mathcal{S} \wedge T.v > 0\ \}) \tag{29}$$

By (20), $L = \max\{\ T.p \mid T \in \mathcal{S} \wedge T.v > 0\ \}$ is the smallest L value at which circumstantial demand no longer exists, *i.e.*, v_L becomes 0.

Removing v_L from (21) and applying Rule 2 produces the upper bound shown below.

$$\frac{\sum_{T\in\mathcal{S}}\left(\frac{T.u\cdot L}{Q}\right)+\beta}{L-\epsilon-2}$$

Reorganizing the terms produces the equivalent form shown below.

$$\frac{1}{Q}\cdot\sum_{T\in\mathcal{S}}T.u+\frac{\frac{1}{Q}\cdot\left(\sum_{T\in\mathcal{S}}T.u\right)\cdot(\epsilon+2)+\beta}{L-\epsilon-2}$$

Applying (9) produces the following equivalent form:

$$\mathcal{S}.I_P+\frac{\mathcal{S}.I_P\cdot(\epsilon+2)+\beta}{L-\epsilon-2}.$$

This leads to the following definition:

$$\phi(\mathcal{S},L)\ \stackrel{\text{def}}{=}\ \mathcal{S}.I_P+\frac{\Psi(\mathcal{S})}{L-\epsilon-2}, \tag{30}$$

where

$$\Psi(\mathcal{S})\ \stackrel{\text{def}}{=}\ \mathcal{S}.I_P\cdot(\epsilon+2)+\beta. \tag{31}$$

Again, the behavior of $\phi(\mathcal{S},L)$ is determined by the characteristic function $\Psi(\mathcal{S})$.

6.2 Termination

We now explain how Property MC is used to detect and to force termination.

Decreasing monotonicity. The following theorem and corollary characterize how decreasing monotonicity allows the unbounded reweighting search space to be truncated (possibly at the cost of increased inflation) by considering a single value of $\phi(\mathcal{S},L)$.

Theorem 6 *If $\phi(\mathcal{S},L)$ is decreasing and $w \geq \phi(\mathcal{S},L)$ for some $L \geq L_\phi$, then*

$$w \geq \max\{\ \Delta(\mathcal{S},L')\ \mid\ L' \geq L\ \}.$$

Proof. Since $\phi(\mathcal{S},L)$ is a decreasing function, Property MC implies that if $w \geq \phi(\mathcal{S},L)$ for some $L \geq L_\phi$, then $w \geq \phi(\mathcal{S},L')$ for all $L' \geq L$. It follows from Property BC that $w \geq \Delta(\mathcal{S},L')$ for all $L' \geq L$. Hence, the theorem holds. $\square$

Corollary 5 *If $\phi(\mathcal{S}, L)$ is decreasing, $w \geq \max\{\ \Delta(\mathcal{S}, L') \mid L > L' \geq L_0\ \}$, and $w \geq \phi(\mathcal{S}, L)$ for some $L \geq L_\phi$, then $w \geq \mathcal{S}.w_{\text{opt}}$.*

Proof. Follows trivially from Theorem 6. □

Non-decreasing monotonicity. The next theorem and corollary consider the case in which $\phi(\mathcal{S}, L)$ is non-decreasing with increasing L. These results are based on the limit of $\phi(\mathcal{S}, L)$ as $L \to \infty$, which is denoted $\mathcal{S}.w_\phi$. By Property MC, which guarantees that $\phi(\mathcal{S}, L)$ does not oscillate, and the fact that $\phi(\mathcal{S}, L)$ cannot approach ∞ in the limit,[7] this limit always exists. When $\phi(\mathcal{S}, L)$ tightly bounds $\Delta(\mathcal{S}, L)$, $\mathcal{S}.w_\phi$ typically equals the ideal weight of the supertask, which is the case when applying Rules 1 and 2 to the example scenarios considered here. For instance, consider (30). As $L \to \infty$, the second term (*i.e.*, $\frac{\Psi(\mathcal{S})}{L-\epsilon-2}$) goes to zero. Hence, $\mathcal{S}.w_\phi = \mathcal{S}.I_P$.

Theorem 7 *If $\phi(\mathcal{S}, L)$ is non-decreasing and $w \geq \mathcal{S}.w_\phi$, then*

$$w \geq \max\{\ \Delta(\mathcal{S}, L') \mid L' \geq L_\phi\ \}.$$

Proof. Since $\phi(\mathcal{S}, L)$ is a non-decreasing function, the value $\mathcal{S}.w_\phi$ upper bounds $\phi(\mathcal{S}, L')$ for all $L' \geq L_\phi$. It follows from Property BC that the theorem holds. □

Corollary 6 *If $\phi(\mathcal{S}, L)$ is non-decreasing, $w \geq \max\{\ \Delta(\mathcal{S}, L') \mid L_\phi > L' \geq L_0\ \}$, and $w \geq \mathcal{S}.w_\phi$, then $w \geq \mathcal{S}.w_{\text{opt}}$.*

Proof. Follows trivially from Theorem 7. □

Ensuring termination. The results presented above demonstrate how comparing a candidate weight to a single value of $\phi(\mathcal{S}, L)$ is sufficient to draw conclusions about an unbounded number of comparisons to $\Delta(\mathcal{S}, L)$ values. Forcing termination is equally trivial: we can ensure that the candidate weight w upper bounds either $\phi(\mathcal{S}, L)$ or $\mathcal{S}.w_\phi$ by assigning $w := \max(w, \phi(\mathcal{S}, L))$ or $w := \max(w, \mathcal{S}.w_\phi)$, respectively. In the next section, we present a general reweighting algorithm based on the properties described in this section.

7 The Reweighting Algorithm

Using the results of the last section, a supertask weight can be selected using the algorithm shown in Figure 6. This algorithm takes five parameters and returns a boolean value. The return value is true if and only if an acceptable weight was found, where "acceptable" is defined by the parameters, as described below.

[7] Such behavior would imply that demand is unbounded. However, unbounded demand cannot be produced by a finite number of tasks.

procedure `Reweight`($\mathcal{S}$, $w_{\min}$, $w_{\max}$, $L_{\max}$, $n_{\max}$) **returns boolean**
$n := 0$;
`InitGenerator`($\mathcal{S}$);
$L :=$ `Generate`();
$\mathcal{S}.w := w_{\min}$;
while $(L < L_\phi) \wedge (\mathcal{S}.w \leq w_{\max})$ **do**
    `CheckWeight`($\mathcal{S}$, L, n)
od;
if $\Psi(\mathcal{S}) \leq 0$ **then**
    $\mathcal{S}.w := \max(\mathcal{S}.w, \mathcal{S}.w_\phi)$
else
    while $(L < L_{\max}) \wedge (n < n_{\max}) \wedge (\mathcal{S}.w < \phi(\mathcal{S}, L)) \wedge (\mathcal{S}.w \leq w_{\max})$ **do**
        `CheckWeight`($\mathcal{S}$, L, n)
    od;
    $\mathcal{S}.w := \max(\mathcal{S}.w, \phi(\mathcal{S}, L))$
fi;
return $(\mathcal{S}.w \leq w_{\max})$

procedure `CheckWeight`($\mathcal{S}$, L, n)
$\mathcal{S}.w := \max(\mathcal{S}.w, \Delta(\mathcal{S}, L))$;
$n := n + 1$;
$L :=$ `Generate`()

Figure 6: Algorithm for selecting a safe supertask weight.

7.1 Parameter Descriptions

The first and most obvious parameter is the supertask $\mathcal{S}$ for which a weight should be selected. $w_{\min}$ and $w_{\max}$ define the range of acceptable supertask weights, *i.e.*, an algorithm invocation returns true if and only if $\mathcal{S}.w \in [w_{\min}, w_{\max}]$ upon termination and $\mathcal{S}.w$ ensures the timeliness of the component tasks. These parameters are assumed to satisfy $0 \leq w_{\min} \leq w_{\max} \leq 1$. Our algorithm is conservative in that a failure (*i.e.*, a return value of *false*) does not preclude the existence of a weight in the range $[w_{\min}, w_{\max}]$ that is capable of ensuring timeliness.

$L_{\max}$ and $n_{\max}$ are the length limit and computation limit, respectively. $L_{\max}$ specifies the largest value of L to consider before forcing the invocation to terminate. Similarly, $n_{\max}$ specifies the maximum number of $\Delta(\mathcal{S}, L)$ values to check before forcing termination. Termination is forced using the approach described in the previous section.

Special cases. In addition to searching for safe supertask weights within a range of values, three special cases of reweighting are common in practice. First, the optimal[8] weight (*i.e.*, $\mathcal{S}.w_{\text{opt}}$) can be sought by invoking `Reweight`($\mathcal{S}$, 0, 1, ∞, ∞). (As discussed below, it may not be possible to identify the optimal weight in bounded time.) Second, the safety of a given weight w can be determined by invoking `Reweight`($\mathcal{S}$, w, w,

[8]This solution is optimal with respect to the presented approach. This weight is *not* guaranteed to be the smallest weight for which timeliness is guaranteed.

∞, ∞). Third, an upper bound on inflation can be computed by the call `Reweight`($\mathcal{S}$, 0, 1, 0, ∞). This call immediately forces termination of line 9. As a result, the return value is the most pessimistic solution that can be generated by `Reweight` and equals $\max(\max\{\ \Delta(\mathcal{S}, L) \mid L_\phi \leq L < L_0\ \}, \phi(\mathcal{S}, L_0))$. When $L_\phi = L_0$, this bound is simply $\phi(\mathcal{S}, L_0)$ and can be computed without invoking `Reweight`.

7.2 Algorithm Description

`Reweight` begins by initializing variables in lines 1–4. In lines 5 and 6, each $L \in \mathcal{L}$ that satisfies $L < L_\phi$ is checked to ensure that $\mathcal{S}.w \geq \max\{\ \Delta(\mathcal{S}, L') \mid L_\phi > L' \geq L_0\ \}$ holds at line 7. The remaining values are then handled by either line 8 or lines 9–11, based on whether $\phi(\mathcal{S}, L)$ is non-decreasing or decreasing, respectively. In the former case, Corollary 6 implies that if $\mathcal{S}.w \geq \mathcal{S}.w_\phi$, then $\mathcal{S}.w$ is guaranteed to be safe. Line 8 ensures that $\mathcal{S}.w \geq \mathcal{S}.w_\phi$ holds. In the latter case, weight restrictions must be checked (lines 9–10) until either a user-provided limit is reached (checked by the first two conditions given in line 9) or an value L is found that satisfies the conditions set forth in Corollary 5 (checked by the third condition given in line 9). In the event that termination is forced, line 11 ensures that the conditions set forth in Corollary 5 still hold and hence that $\mathcal{S}.w$ is a safe weight. (Note that if the **while** loop terminates due to the third condition in line 9, then line 11 has no effect.) Line 12 then reports whether the invocation was successful (see below).

`CheckWeight` is invoked to compare the current supertask weight to the $\Delta(\mathcal{S}, L)$ restriction, which is done in line 13. Line 14 then updates the computation counter to reflect the comparison. Finally, L is advanced to the next highest value of $L \in \mathcal{L}$ in line 15.

Correctness of the return value. An important property of `Reweight` is that $\mathcal{S}.w$ is non-decreasing after line 4, as implied by lines 8, 11, and 13. As a result, both $\mathcal{S}.w \geq w_{\min}$ and $\mathcal{S}.w > w_{\max}$ are invariant once established. Since line 4 establishes the $\mathcal{S}.w \geq w_{\min}$, it follows that $\mathcal{S}.w \leq w_{\max} \Leftrightarrow \mathcal{S}.w \in [w_{\min}, w_{\max}]$ holds at line 12. To avoid unnecessary computation, both loops terminate immediately if failure ($\mathcal{S}.w > w_{\max}$) is detected (see lines 5 and 9).

7.3 Properties

We now state and prove basic properties of `Reweight`. First, a lower bound on $\mathcal{S}.w$ following the successful completion of a `Reweight` invocation is proved in the theorem given below. This theorem also establishes a lower bound on the inflation produced when reweighting a supertask using the approach described here.

Theorem 8 *If an invocation of* `Reweight` *on* $\mathcal{S}$ *terminates, then upon termination,* $\mathcal{S}.w \geq \mathcal{S}.w_\phi$ *holds if* $\Psi(\mathcal{S}) \leq 0$ *and* $\mathcal{S}.w > \mathcal{S}.w_\phi$ *holds if* $\Psi(\mathcal{S}) > 0$.

Proof. The $\Psi(\mathcal{S}) \leq 0$ case follows trivially from line 8 of `Reweight`. When $\Psi(\mathcal{S}) > 0$, $\phi(\mathcal{S}, L)$ approaches $\mathcal{S}.w_\phi$ from above in the limit. Hence, $\phi(\mathcal{S}, L) > \mathcal{S}.w_\phi$ holds for all L. Thus, $\mathcal{S}.w > \mathcal{S}.w_\phi$ holds after the eventual execution of line 11. It follows that $\mathcal{S}.w > \mathcal{S}.w_\phi$ holds upon termination also. □

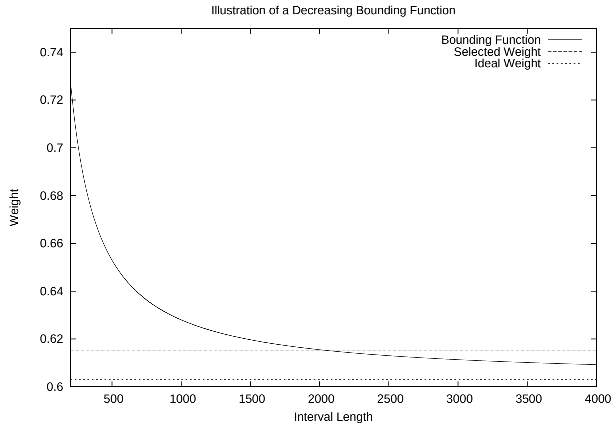


Figure 7: When $\phi(\mathcal{S}, L)$ is a decreasing function, the value of the function will eventually drop below any weight in the range $(\mathcal{S}.w_\phi, 1]$. In the graph, the "Bounding Function" and "Ideal Weight" curves correspond to $\phi(\mathcal{S}, L)$ and $\mathcal{S}.w_\phi$, respectively.

Theorem 8 raises the question of whether termination is guaranteed. When $\Psi(\mathcal{S}) \leq 0$ holds, termination obviously occurs. However, termination may not occur when $\Psi(\mathcal{S}) > 0$ holds. The following lemma, theorem, and corollary characterize the circumstances under which termination does not occur.

Lemma 2 *If $\mathcal{S}.w > \mathcal{S}.w_\phi$ is established during an invocation of* `Reweight`*, then termination is guaranteed.*

Proof. Termination can only be avoided by taking the code branch leading to line 9,[9] which only occurs when $\Psi(\mathcal{S}) > 0$. In this case, $\phi(\mathcal{S}, L)$ decreases as L increases. Because $\phi(\mathcal{S}, L)$ approaches $\mathcal{S}.w_\phi$ in the limit, the value of $\phi(\mathcal{S}, L)$ will eventually drop (and remain) below any weight in the range $(\mathcal{S}.w_\phi, 1]$.

Figure 7 illustrates this property. In this example, $\mathcal{S}.w_\phi = 0.603$. Suppose that $\mathcal{S}.w = 0.615$ (> 0.603) is established during an invocation of `Reweight`. As shown, the value of $\phi(\mathcal{S}, L)$ drops below $\mathcal{S}.w$ around $L = 2200$. Hence, $\mathcal{S}.w \geq \phi(\mathcal{S}, L)$ holds for all $L \geq 2200$ since $\phi(\mathcal{S}, L)$ is a decreasing function. It follows that the loop at line 9 must eventually terminate due to its third condition. The lemma follows. $\square$

Theorem 9 *An invocation of* `Reweight` *on $\mathcal{S}$ does not terminate if and only if all of the following conditions hold:*

1. $\Psi(\mathcal{S}) > 0$;
2. $w_{\min} \leq \mathcal{S}.w_\phi$;
3. $\mathcal{S}.w_{\text{opt}} \leq \mathcal{S}.w_\phi$;
4. $\mathcal{S}.w_{\text{opt}} \leq w_{\max}$;

[9]Since L_ϕ is required to be finite, the loop at line 5 must eventually terminate due to its first condition.

5. $L_{\max} = n_{\max} = \infty$.

Proof. By the algorithm's code listing, termination can be avoided only by becoming trapped in the **while** loop at line 9. To prove sufficiency, we show that the conditions given above guarantee that the loop is reached and that the four conditions in line 9 always hold (and hence that the loop never terminates). Condition 1 is sufficient to ensure that the loop is reached via line 7. Now consider the four conditions in line 9. By Condition 5, the first two conditions in line 9 always hold. It remains to show that the third and fourth conditions always hold.

To show that the third condition in line 9 holds, we show that $\mathcal{S}.w \leq \mathcal{S}.w_\phi$ is invariant after the execution of line 4. We begin by explaining why $\mathcal{S}.w \leq \mathcal{S}.w_\phi \Rightarrow \mathcal{S}.w < \phi(\mathcal{S}, L)$. By Condition 1, $\phi(\mathcal{S}, L)$ is decreasing. Hence, $\phi(\mathcal{S}, L)$ approaches (but never equals) $\mathcal{S}.w_\phi$ in the limit. For instance, consider the definition of $\phi(\mathcal{S}, L)$ given in (27). In this case, $\mathcal{S}.w_\phi = \mathcal{S}.I_Q$ and $\phi(\mathcal{S}, L) = \mathcal{S}.w_\phi + \frac{\Psi(\mathcal{S})}{L-\epsilon}$. The term $\frac{\Psi(\mathcal{S})}{L-\epsilon}$ approaches (but never equals) zero as $L \to \infty$. Hence, if $\mathcal{S}.w \leq \mathcal{S}.w_\phi$ is invariant, then it follows that $\mathcal{S}.w < \phi(\mathcal{S}, L)$ holds for all L (and that the third condition in line 9 always holds).

We now prove that $\mathcal{S}.w \leq \mathcal{S}.w_\phi$ is invariant after line 4. It follows from Condition 2 that $\mathcal{S}.w \leq \mathcal{S}.w_\phi$ holds immediately after line 4. By (16), $\Delta(\mathcal{S}, L) \leq \mathcal{S}.w_{\text{opt}}$ for all L. Hence, by Condition 3, $\mathcal{S}.w \leq \mathcal{S}.w_\phi$ holds after each execution of line 13. Hence, $\mathcal{S}.w \leq \mathcal{S}.w_\phi$ is invariant after line 4.

Similarly, we show that the fourth condition in line 9 always holds by showing that $\mathcal{S}.w \leq w_{\max}$ is invariant after line 4. By Condition 2 and the property $w_{\min} \leq w_{\max}$, $\mathcal{S}.w \leq w_{\max}$ holds immediately after line 4. By Condition 4, $\mathcal{S}.w \leq w_{\max}$ holds after each execution of line 13. Hence, $\mathcal{S}.w \leq w_{\max}$ is invariant after line 4. This completes the proof of sufficiency.

Now, consider necessity. If Condition 1 does not hold, then the loop in line 9 is never executed and hence cannot prevent termination. We now argue that negating each of the remaining conditions guarantees termination. First, if Condition 5 does not hold, then one of the first two conditions in line 9 will eventually cause termination. If Condition 4 does not hold, then $\mathcal{S}.w > w_{\max}$ is eventually established by line 13 (unless it is established sooner by line 4), which results in termination of the loop by its fourth condition. Similarly, if either Condition 2 or 3 does not hold, then $\mathcal{S}.w > \mathcal{S}.w_\phi$ is eventually established by either line 4 or 13, respectively. By Lemma 2, termination is ensured once $\mathcal{S}.w > \mathcal{S}.w_\phi$ is established. This completes the proof of necessity. $\square$

Corollary 7 *If* `Reweight` *is invoked on $\mathcal{S}$, then termination is guaranteed whenever either $w_{\min} > \mathcal{S}.w_\phi$ or $\Psi(\mathcal{S}) \leq 0$ holds.*

Proof. Follows trivially from Theorem 9. $\square$

7.4 Balancing Speed and Inflation

`Reweight` provides three methods for controlling the speed-versus-accuracy trade-off, each of which involves the use of one of $w_{\min}$, $L_{\max}$, or $n_{\max}$. Each method discussed here provides a predictable lower bound on reweighting overhead and an upper bound on the amount of computation performed. For the initial description of each method (given below), we assume that only one of $w_{\min}$, $L_{\max}$, or $n_{\max}$ is set to a non-trivial value.[10] When two or more parameters are used simultaneously, the speed and accuracy of the process is determined by the parameter that implies the smallest number of computations.

First, the reweighting process can be seeded with a non-ideal minimum weight, *i.e.*, $w_{\min}$ can be assigned a value larger than $\mathcal{S}.w_\phi$. The benefit of such an assignment can be seen in Figure 7. Specifically, $\phi(\mathcal{S}, L)$ crosses larger weights at lower values of L. It is straightforward to calculate the smallest L value for which $\phi(\mathcal{S}, L)$ is less than $w_{\min}$ (and hence, for which the third condition at line 9 is guaranteed to hold). Let L_{last} denote this derived value. It follows that using $w_{\min}$ produces the same result as passing L_{last} as the $L_{\max}$ parameter.

Second, the $L_{\max}$ parameter can be used to bound the search space of the reweighting process. As in the previous case, the $w_{\min}$ parameter can be used to achieve the same result. Specifically, the use of $L_{\max}$ achieves the same result as passing $\phi(\mathcal{S}, L_{\max})$ as the $w_{\min}$ parameter. Hence, the accuracy and speed bounds are computed as in the previous case.

Finally, the $n_{\max}$ parameter can be used to bound the number of $\Delta(\mathcal{S}, L)$ computations. Unfortunately, it is difficult to characterize how $n_{\max}$ relates to using the $L_{\max}$ parameter due to the fact that only a subset of the L values are actually checked. Under the pessimistic assumption that every L value is tested, $L_{last} = L_0 + n_{\max} - 1$. (Here, we assume that termination is forced.) Using this relationship, $n_{\max}$ can be related to both $L_{\max}$ and $w_{\min}$ as described above.

8 Experimental Results

In this section, we present the results of an experimental evaluation of `Reweight`. This evaluation consisted of two studies. Due to length considerations, only a small subset of the results produced by this evaluation are presented here. The complete results can be found in (Holman, 2004).

The first study focused on the reweighting overhead produced by scheduling periodic tasks under each of the example scenarios. For the QB-EPDF scenario, task parameters had to be mapped to weights first. The mapping rules for periodic and sporadic tasks can be found in (Holman, 2004). An FP-EDF scenario was actually tested instead of the FP-EDF-NP scenario. As explained earlier in the paper, scheduling without non-preemptable code segments can be considered by simply setting the maximum duration of such segments to zero in the equations presented earlier.

[10] $L_{\max}$ and $n_{\max}$ are set to ∞ when not used; $w_{\min}$ can be set to any value in $[0, \mathcal{S}.w_\phi]$.

The second study focused on the speed-versus-accuracy trade-off. Again, both the QB-EPDF and FP-EDF scenarios were considered. The primary goal of this study was to approximate the relationship between inflation and the amount of computation performed. Specifically, the objective was to determine the number of computations that must be performed to produce a solution that is approximately optimal.

8.1 Study 1: Reweighting Overhead

In this section, we present the details and results of the first study.

Sampling. Both studies presented in this section are based on comparing randomly generated component task sets uniformly drawn from a sample space. For this study, the component-task count was chosen from the range $2, \ldots, 40$. The total utilization of all component tasks was chosen from $0.02, \ldots, 0.9$. Finally, task periods were chosen from $5, \ldots, 5,000$. To ensure that the random samples provided reasonable coverage of this sample space, the task count, total utilization, and minimum period were systematically chosen so that coverage of the sample space (with respect to these parameters) would be uniform.

Measurement. The following measurements were taken during the experimental runs.

No Supertasks: a baseline measurement of the total weight of all tasks before reweighting; this measurement reflects the mapping overhead produced when assigning weights to the periodic tasks.

QB-EPDF: the weight assigned to the supertask under the QB-EPDF scenario; this measurement reflects both mapping and reweighting overhead.

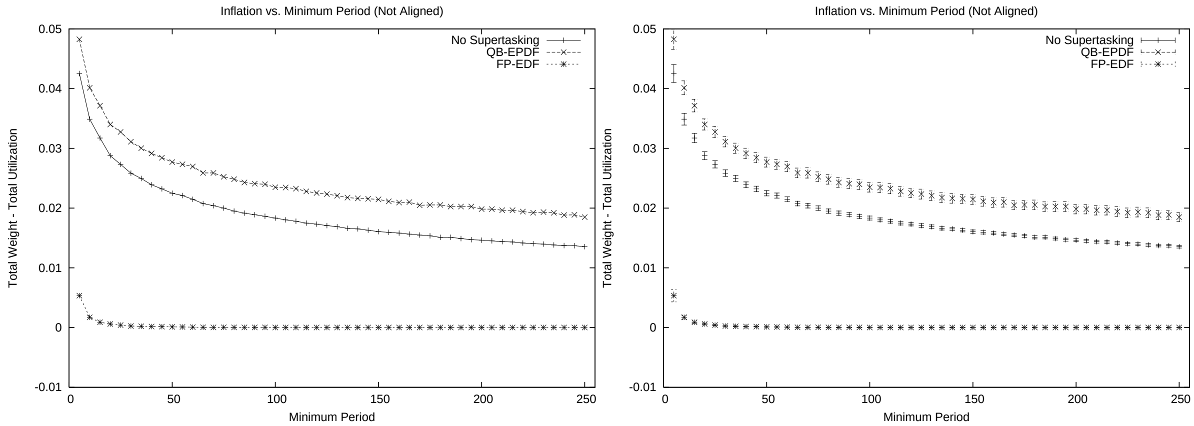
FP-EDF: the weight assigned to the supertask under the FP-EDF scenario; this measurement reflects reweighting overhead alone since fully preemptive supertasking does not require parameter mapping.

To ensure termination, $\mathcal{S}.w_\phi + 10^{-5}$ was passed as $w_{\min}$ when invoking `Reweight`. The quantity 10^{-5} was used because we consider such a small degree of inflation to be negligible. Only samples which were schedulable under all approaches were considered. We refer to such samples as *valid*. When invalid samples were generated, they were simply discarded and replaced by a new sample.

Results. We present only the plots of inflation versus the minimum period here, which are shown in Figure 8. Our goal is only to demonstrate the behavior of inflation under each approach by presenting a representative sample of the results. The full results can be found in (Holman, 2004).

As shown, the FP-EDF scenario produces very little reweighting overhead. Indeed, the overhead appears to be negligible on average for period sizes of twenty or more. The QB-EPDF scenario performs worse, but still sacrifices only around 0.005 to reweighting overhead.[11] The difference in overhead is due to the fact that

[11] The reweighting overhead under the QB-EPDF scenario corresponds to the difference between the QB-EPDF and "No Supertasks" measurements.



(a) Sample Means **(b) 99% Confidence Intervals**

Figure 8: Plots show overhead under both the QB-EPDF and FP-EDF scenarios with scheduling periodic and sporadic tasks. The "No Supertasks" measurement shows the mapping overhead produced by assigning weights to the tasks to enable use of the QB-EPDF scenario. The figure shows **(a)** the mean overhead, and **(b)** the 99% confidence interval associated with each sample mean.

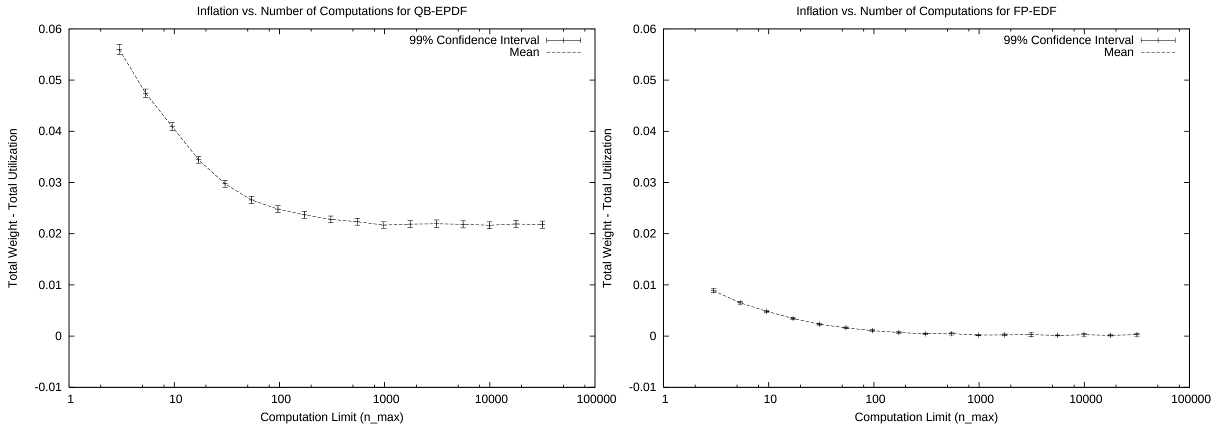
tasks are required to execute at approximately steady rates under EPDF. Hence, scheduling is inherently more difficult, and hence more costly.

These plots suggest that both reweighting and mapping overhead are inversely proportional to the minimum task period. Since our focus in this paper is only reweighting overhead, we ignore the mapping overhead here. (See (Holman, 2004) for a discussion of mapping overhead.) Indeed, this relationship is not surprising. A simple upper bound on inflation is given by $\phi(\mathcal{S}, L_0)$. As demonstrated by (27) and (30), $\phi(\mathcal{S}, L) \approx I + \frac{\Psi(\mathcal{S})}{L}$ where I denotes the ideal weight of the supertask (*i.e.*, either $\mathcal{S}.I_Q$ or $\mathcal{S}.I_P$). Hence, reweighting overhead is bounded (approximately) by $\frac{\Psi(\mathcal{S})}{L_0}$. By (13), L_0 scales directly with periods when scheduling jobs. By (12), L_0 scales with windows sizes under EPDF scheduling. Window sizes decrease with increasing weight. Using the mapping rules in (Holman, 2004), weights tend to increase as periods decrease. Hence, L_0 scales proportionally to periods under EPDF as well. As a result, the worst-case and average-case reweighting overhead tends to decrease as periods are increased.

8.2 Study 2: Speed Versus Accuracy

In this section, we present the details and results of the second study, which focused on the speed-versus-accuracy trade-off. Specifically, two experiments were conducted. In the first experiment, the value of $n_{\max}$ was varied and the impact on the inflation was observed. In the second experiment, the value of $w_{\min}$ was varied and the impact on the number of computations performed was observed. Since the impact of $L_{\max}$ depends on the task periods (due to $\mathcal{L}$), the impact of setting this parameter to a specific value will certainly not impact all task sets equally. Hence, this parameter was not considered in this experiment.

Sampling. In the first experiment, $n_{\max}$ was systematically varied across the range $1, \ldots, 100,000$. In



(a) QB-EPDF **(b) FP-EDF**

Figure 9: Plots show the impact of $n_{\max}$ on the reweighting overhead under each of the **(a)** QB-EPDF and **(b)** FP-EDF scenarios.

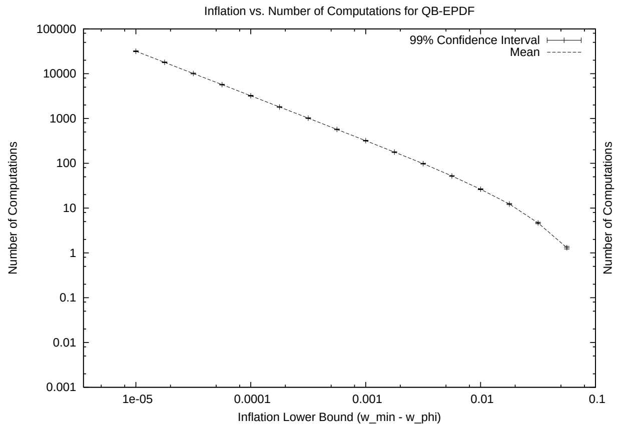
the second experiment, the difference $w_{\min} - \mathcal{S}.w_\phi$ was systematically varied over the range $0.00001, \ldots, 0.1$. In both experiments, the task count was chosen from the ranges $2, \ldots, 40$, the total utilization was chosen from the ranges $0.02, \ldots, 0.9$, and the task periods were chosen from the ranges $5, \ldots, 2000$. Again, the task count, total utilization, and minimum task period were systematically varied to ensure uniform sampling.

Per-computation execution time. To place the estimates of computation in perspective, we ran the reweighting algorithm on a 1.4 GHz Pentium 4 desktop computer and measured the resulting mean per-computation execution time, *i.e.*, the total execution time divided by the value of n upon termination of the algorithm. Since several computations within the algorithm have time complexity $O(|\mathcal{S}|)$, we varied $|\mathcal{S}|$ to determine how the execution time scales. The collected data suggests that the per-computation execution time in microseconds is approximately $1.32 + 0.1357 \cdot |\mathcal{S}|$, *i.e.*, on the order of a few microseconds. Hence, a single task set can be reweighted in at most a few seconds, even when high precision is desired.

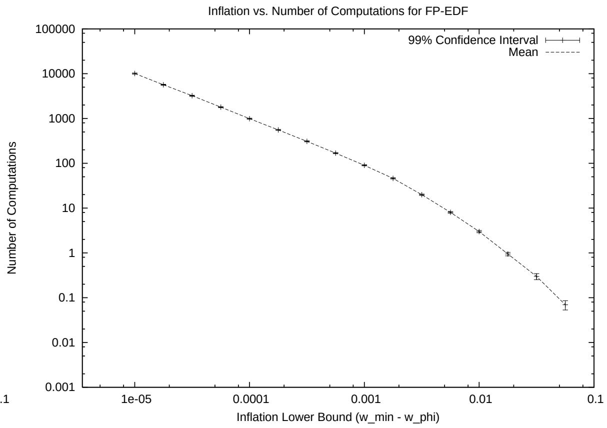
Using $n_{\max}$. Figure 9 shows the impact of $n_{\max}$ under both the QB-EPDF and FP-EDF scenarios. Notice that both graphs are shown in log scale. As shown in Figure 9, the mean inflation produced by reweighting stabilizes at approximately $n_{\max} = 1000$ under each scenario. However, the mean of the FP-EDF scenario, shown in Figure 9(b), is reasonably stable much earlier, *i.e.*, around $n_{\max} = 100$.

Using $w_{\min}$. Figure 10 shows the impact of using an inflated $w_{\min}$ value under the QB-EPDF and FP-EDF scenarios. Again, both plots are shown in log scale. Surprisingly, the relationship between $w_{\min} - \mathcal{S}.w_\phi$ and the number of computations performed appears to be approximately linear[12] throughout the graph. By extrapolation, the line segment over the x range $[0.00001, 0.1]$ for QB-EPDF (respectively, over the x range $[0.00001, 0.001]$ for FP-EDF) appears to fit the equation $y = 10^{-k}\frac{1}{x}$, where $k = 0.5$ (respectively, $k = 1$).

[12]By linear, we mean only that the graph depicts a straight line. Because the graph is presented in log scale, this does not suggest that the relationship can be expressed as a linear equation.



(a) QB-EPDF



(b) FP-EDF

Figure 10: Plots show the impact of $w_{\min}$ on the number of computations performed under each of the **(a)** QB-EPDF and **(b)** FP-EDF scenarios.

The cause of this behavior is not immediately obvious; further study is needed to better understand the relationship between these quantities. For larger $w_{\min}$ values, this relationship does not hold. We speculate that the reason for this is that the $w_{\min}$ parameter is set so high that $\phi(\mathcal{S}, L_0) > w_{\min}$ holds initially. When this happens, the invocation terminates after performing only minimal computation.

9 Conclusion

In this paper, we have presented a general framework for assigning supertask weights so that the timing constraints of component tasks are guaranteed. This framework consists of a combination of analysis and a weight-selection algorithm. The primary limitation of this approach is that reweighting may require considerable (possibly even unbounded) computation. To address this concern, we presented a necessary and sufficient condition for termination. In addition, we supplemented the reweighting algorithm with parameters that allow a user to force early termination after a specified amount of computation, at the expense of some additional weight inflation. To demonstrate the use of this framework, we considered one quantum-based and one fully preemptive scheduling scenario. We concluded by presenting a subset of the results from an experimental evaluation of the proposed reweighting algorithm. This evaluation suggests that weight inflation due to reweighting should be reasonably low in most cases. Hence, the proposed technique is a practical means to implementing hierarchal scheduling in real multiprocessor systems.

Due to length considerations, we have not presented all results relating to this work. A more complete coverage of this topic can be found in Holman (2004). Additional results not presented here include a slightly more general analytical framework, additional examples, the full results of the experimental evaluation, and a discussion of how supertasks can be used as an optimization tool.

Practical considerations. As with any analysis, pessimism can arise from a variety of sources when using the proposed framework. We divide this pessimism into two general categories: user-introduced pessimism and inherent pessimism.

User-introduced pessimism results from the use of poor-quality bounds and estimates when defining the scenario-specific parts of the framework (*e.g.*, supply and demand formulas). To ensure high-quality results, such formulas should always be justified by additional analysis and proofs, as was done for the examples in this paper. One potential source of user-introduced pessimism that may not be immediately obvious is $\phi(\mathcal{S}, L)$. By Corollary 7, $w_{\min} > \mathcal{S}.w_\phi$ is needed to ensure termination of the reweighting algorithm. Since $\mathcal{S}.w_\phi$ is the limit of $\phi(\mathcal{S}, L)$ as $L \to \infty$, using a loose bound for $\phi(\mathcal{S}, L)$ can result in an unnecessarily large $\mathcal{S}.w_\phi$ value, and hence in additional inflation.

Inherent pessimism results from the derivations and general approach taken by the framework. The only step of the framework that has the potential to introduce pessimism is the derivation of $\Delta(\mathcal{S}, L)$ (see Section 5.1). In order to make the reweighting computation tractable, the focus of the analysis shifts at this point from considering the supply and demand over a specific interval $[t, t+L)$ to the *minimum* supply and *maximum* demand over *any* interval of length L. However, unless the minimum supply and maximum demand can actually occur within the same interval, this shift will result in some additional pessimism. In addition, when manipulating the schedulability condition to isolate the $\mathcal{S}.w$ term, it may be necessary to relax some bounds slightly, as demonstrated by the derivation of $\Delta(\mathcal{S}, L)$ for the FP-EDF-NP scenario.

Future work. Many interesting aspects of supertasking remain uninvestigated, of which three are most prominent. First, the approach presented in this paper considers only the use of servers based on the Pfair task model. Servers based on other task models, such as the ERfair model proposed by Anderson and Srinivasan Anderson and Srinivasan (2000), may exhibit less weight inflation and provide other interesting properties.

Second, the problem of assigning tasks to supertasks in order to minimize the total system overhead has not been investigated in detail. (We briefly outlined an algorithm for this problem when discussing the partitioning approach in Section 3.) In prior work, we have considered a few assignment heuristics for reducing synchronization overhead Holman and Anderson (2002a,b). (Since the task-assignment problem is a more complex form of the bin-packing problem, which is known to be intractable, we consider only the use of heuristics.) However, such simple heuristics will likely not be effective for real systems, in which there are typically several significant forms of overhead (*e.g.*, interrupt handling, scheduling, interprocessor communication, resource contention, cache misses, hardware contention, *etc.*). This problem is further complicated by the fact that the relationship between supertask assignments and the different forms of overhead present in real systems is often difficult to quantify. For instance, by constraining which processors a supertask may execute upon in a bus-based multiprocessor, the worst-case volume of bus traffic (and hence the worst-case contention for the shared bus) can be reduced. However, the relationship between overhead stemming from such bus contention and supertask assignments may be difficult to express in an optimization problem.

Finally, the analysis presented here does not allow for transient periods of overload. Indeed, no prior research has investigated the behavior of Pfair scheduling and its variants under overload conditions. Due to the fair allocation of processor time, Pfair scheduling and variants should continue to behave in a predictable manner even under such conditions.

Acknowledgement: We are grateful to the anonymous reviewers for their helpful suggestions regarding an earlier draft of this paper.

References

L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in Proceedings of the 19th IEEE Real-time Systems Symposium, December 1998.

J. Anderson and A. Srinivasan, "Pfair scheduling: Beyond periodic task systems," in Proceedings of the Seventh International Conference on Real-time Computing Systems and Applications, pp. 297–306, December 2000.

J. Anderson and A. Srinivasan, "Mixed Pfair/ERfair scheduling of asynchronous periodic tasks," in Proceedings of the 13th Euromicro Conference on Real-time Systems, pp. 76–85, June 2001.

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

S. Baruah, J. Gehrke, and C. G. Plaxton, "Fast scheduling of periodic tasks on multiple resources," in Proceedings of the 9th International Parallel Processing Symposium, pp. 280–288, April 1995.

G. Bollella *et al.*, The Real-time Specification for Java, Addison Wesley, 2000.

M. Caccamo and L. Sha, "Aperiodic servers with resource constraints," in Proceedings of the 22nd IEEE Real-time Systems Symposium, pp. 161–170. December 2001.

X. Feng and A. Mok, "A model of hierarchical real-time virtual resources," in Proceedings of the 23rd IEEE Real-time Systems Symposium, pp. 26–35, December 2002.

P. Holman, "On the implementation of Pfair-scheduled multiprocessor systems," Ph.D. Thesis, University of North Carolina at Chapel Hill, 2004.

P. Holman and J. Anderson, "Guaranteeing Pfair supertasks by reweighting," in Proceedings of the 22nd IEEE Real-time Systems Symposium, pp. 203–212. December 2001.

P. Holman and J. Anderson, "Object sharing in Pfair-scheduled multiprocessor systems," in Proceedings of the 14th Euromicro Conference on Real-time Systems, pp. 111-120, June 2002.

P. Holman and J. Anderson, "Locking in Pfair-scheduled multiprocessor systems," in Proceedings of the 23rd IEEE Real-time Systems Symposium, pp. 149–158, December 2002.

P. Holman and J. Anderson, “Using hierarchal scheduling to improve resource utilization in multiprocessor real-time systems,” in Proceedings of the 15th Euromicro Conference on Real-time Systems, pp. 41–50, July 2003.

G. Lamastra, G. Lipari, and Luca Abeni, “A bandwidth inheritance algorithm for real-time task synchronization in open systems,” in Proceedings of the 22nd IEEE Real-time Systems Symposium, pp. 151–160. December 2001.

G. Lipari and E. Bini, “Resource partitioning among real-time applications,” in Proceedings of the 15th Euromicro Conference on Real-time Systems, pp. 35–43, July 2003.

C. Liu and J. Layland, “Scheduling algorithms for multiprogramming in a hard real–time environment,” Journal of the ACM, 30:46–61, January 1973.

M. Moir and S. Ramamurthy, “Pfair scheduling of fixed and migrating periodic tasks on multiple resources,” in Proceedings of the Twentieth IEEE Real-time Systems Symposium, pp. 294–303, December 1999.

A. Mok, “Fundamental design problems for the hard real-time environment,” Ph.D. Thesis, Massachussetts Institute of Technology, 1983.

A. Mok and X. Feng, “Towards compositionality in real-time resource partitioning based on regularity bounds,” in Proceedings of the 22nd IEEE Real-time Systems Symposium, pp. 128–138, December 2001.

A. Mok, X. Feng, and D. Chen, “Resource partition for real-time systems,” in Proceedings of the Real-time Technology and Applications Symposium, pp. 75–84, May 2001.

I. Shin and I. Lee, “Periodic resource model for compositional real-time guarantees,” in Proceedings of the 24th IEEE Real-time Systems Symposium, pp. 2–13, December 2003.

A. Srinivasan and J. Anderson, “Efficient scheduling of soft real-time applications on multiprocessors,” in Proceedings of the 15th Euromicro Conference on Real-time Systems, pp. 51–59, June 2003.

A. Tucker and A. Gupta, “Process control and scheduling issues for multiprogrammed shared-memory multiprocessors,” in Proceedings of the Twelfth ACM Symposium on Operating Systems Principles, pp. 159–166, 1989.

A Summary of Notation

Symbol	Meaning
$\mathcal{N}$	Set of all natural numbers
$\mathcal{Z}$	Set of all integers
Q	Quantum size
M	Processor count
τ	Set of tasks
$\|S\|$	Size of the set S
$\tau.u$	Cumulative utilization of all tasks in τ
J, K	Jobs
T, U	Tasks
$T.w$	Weight of a task T
T_i	The i^{th} subtask of a task T
$\omega(T_i)$	The window associated with the i^{th} subtask of a task T
$r(T_i)$	The pseudo-release of the i^{th} subtask of a task T
$d(T_i)$	The pseudo-deadline of the i^{th} subtask of a task T
$\mathbf{PF}(w)$	A Pfair task with weight w
$\mathit{fluid}(T, t_1, t_2)$	Share of processor time entitled to task T over the interval $[t_1, t_2)$
$\mathit{lag}(T, t)$	The lag of task T at time t
$\mathit{received}(T, t_1, t_2)$	Processor time allocated to task T over the interval $[t_1, t_2)$
β_+	Upper lag scaler
β_-	Lower lag scaler
β	Sum of the lag scalers
ϵ_r	Pseudo-release extension
ϵ_d	Pseudo-deadline extension
ϵ	Sum of window extensions
$T.\phi$	Offset of a periodic or sporadic task T
$T.p$	Period of a periodic or sporadic task T
$T.e$	Per-job execution requirement of a periodic or sporadic task T
$T.d$	Per-job relative deadline of a periodic or sporadic task T
$T.u$	Utilization of a periodic or sporadic task T
$\mathbf{P}(\phi, e, p, d)$	A periodic task with an offset of ϕ, a per-job execution requirement of e, a period of p, and a per-job relative deadline of d
$\mathbf{S}(\phi, e, p, d)$	A sporadic task with an offset of ϕ, a per-job execution requirement of e, a period of p, and a per-job relative deadline of d

Symbol	Meaning
$\mathcal{S}, \mathcal{T}$	Supertasks
$\mathcal{S}.I_Q$	Ideal weight of a supertask using quantum-based scheduling
$\mathcal{S}.I_P$	Ideal weight of a supertask $\mathcal{S}$ using fully preemptive scheduling
$demand(S, t_1, t_2)$	The minimum amount of processor time needed by tasks in S to avoid a timing violation over the interval $[t_1, t_2)$
$supply(S, t_1, t_2)$	The amount of processor time provided to tasks in S over the interval $[t_1, t_2)$
$demand_M(S, t_1, t_2)$	The minimum amount of processor time needed by mandatory requests in S to avoid a timing violation over the interval $[t_1, t_2)$
$demand_C(S, t_1, t_2)$	The minimum amount of processor time needed by requests in S that are not mandatory to avoid a timing violation over the interval $[t_1, t_2)$
L	Interval length
L_0	The shortest interval length over which a request of a component task can arrive and experience a timing violation
$\Delta(\mathcal{S}, L)$	The minimum value of $\mathcal{S}.w$ needed to avoid timing violations over intervals of length L
$\mathcal{L}$	The testing set of $\Delta(\mathcal{S}, L)$
$\mathcal{S}.w_{\text{opt}}$	The smallest value of $\mathcal{S}.w$ that satisfies all $\Delta(\mathcal{S}, L)$ restrictions, *i.e.*, the optimal reweighting solution
υ_L	The maximum circumstantial demand generated by non-preemptable code segments over an interval of length L
$\phi(\mathcal{S}, L)$	The monotonic bounding function of $\Delta(\mathcal{S}, L)$
$\Psi(\mathcal{S})$	The characteristic function of $\phi(\mathcal{S}, L)$
$\mathcal{S}.w_\phi$	The limit of $\phi(\mathcal{S}, L)$ as $L \rightarrow \infty$
L_ϕ	The activation point of $\phi(\mathcal{S}, L)$
$T.\upsilon$	Maximum duration for which task T can execute non-preemptably

B Proofs

In this section, we derive the basic properties of scheduling summarized earlier in Sections 2 and 4.

Window placement. Theorem 1 is established by a trivial extension of the lemma shown below.

Lemma 3 *The following formulas define the placement of relaxed windows:*

$$r(T_i) = \left\lfloor \frac{i - \beta_+}{T.w} \right\rfloor \qquad\qquad d(T_i) = \left\lceil \frac{(i-1) + \beta_-}{T.w} \right\rceil .$$

Proof. These formulas follow directly from (5), which implies that the release and deadline of a subtask T_i are defined as follows:

$$r(T_i) = \min\{\ k \mid k \in \mathcal{Z} \wedge i \cdot Q - \mathit{fluid}(T, 0, k+1) < Q \cdot \beta_+ \ \};\ \text{and}$$
$$d(T_i) = \min\{\ k \mid k \in \mathcal{Z} \wedge (i-1) \cdot Q - \mathit{fluid}(T, 0, k) \leq -Q \cdot \beta_- \ \}.$$

In the above formulas, $\mathcal{Z}$ denotes the set of all integers. Informally, the $r(T_i)$ constraint identifies the earliest slot (k) such that the upper lag constraint ($Q \cdot \beta_+$) is not violated when the i^{th} quanta is received in that slot (*i.e.*, in the interval $[k, k+1)$). The subtask release corresponds to the start of this slot, *i.e.*, time k.[13] On the other hand, the $d(T_i)$ constraint identifies the earliest time k such that the lower lag bound ($-Q \cdot \beta_-$) *is* violated when only $i-1$ quanta are received by k. It follows that the i^{th} quantum must be received in the interval $[k-1, k)$, at the latest. Hence, the subtask deadline occurs at time k.

Applying (2) and rearranging terms to isolate k produces the following equivalent forms.

$$r(T_i) = \min\left\{\ k \ \middle|\ k \in \mathcal{Z} \wedge k > \frac{i - \beta_+}{T.w} - 1 \ \right\}$$
$$d(T_i) = \min\left\{\ k \ \middle|\ k \in \mathcal{Z} \wedge k \geq \frac{(i-1) + \beta_-}{T.w} \ \right\}$$

The lemma follows. □

Window span. Lemma 1, shown below, bounds the number of slots spanned by a sequence of n consecutive windows.

Lemma 1 *Every sequence of consecutive subtasks $T_{i+1}, \ldots, T_{i+n}$ satisfies the following:*

$$\left\lceil \frac{n + \beta - 2}{T.w} \right\rceil + \epsilon \leq d(T_{i+n}) - r(T_{i+1}) \leq \left\lceil \frac{n + \beta - 2}{T.w} \right\rceil + \epsilon + 1$$

Proof. The following derivation establishes the upper bound in the first claim.

[13] Notice that the release time may be negative. It is important to understand that time 0 is simply a reference point that records when scheduling begins. Since no scheduling occurs prior to this point, windows with negative release times are effectively truncated to begin at time 0.

$$
\begin{aligned}
&d(T_{i+n}) - r(T_{i+1}) \\
&= \left(\left\lceil \tfrac{(i+n-1)+\beta_-}{T.w} \right\rceil + \epsilon_d\right) - \left(\left\lfloor \tfrac{(i+1)-\beta_+}{T.w} \right\rfloor - \epsilon_r\right) && \text{, by Theorem 1} \\
&\leq \left\lceil \tfrac{n+\beta_+ +\beta_- -2}{T.w} \right\rceil + \left\lfloor \tfrac{i+1-\beta_+}{T.w} \right\rfloor + 1 - \left\lfloor \tfrac{i+1-\beta_+}{T.w} \right\rfloor + \epsilon_r + \epsilon_d && \text{, } \lceil a+b \rceil \leq \lceil a \rceil + \lfloor b \rfloor + 1 \\
&= \left\lceil \tfrac{n+\beta_+ +\beta_- -2}{T.w} \right\rceil + \epsilon_r + \epsilon_d + 1 && \text{, simplification} \\
&= \left\lceil \tfrac{n+\beta-2}{T.w} \right\rceil + \epsilon + 1 && \text{, by (6) and (7)}
\end{aligned}
$$

The following derivation establishes the lower bound in the first claim.

$$
\begin{aligned}
&d(T_{i+n}) - r(T_{i+1}) \\
&= \left(\left\lceil \tfrac{(i+n-1)+\beta_-}{T.w} \right\rceil + \epsilon_d\right) - \left(\left\lfloor \tfrac{(i+1)-\beta_+}{T.w} \right\rfloor - \epsilon_r\right) && \text{, by Theorem 1} \\
&\geq \left\lceil \tfrac{n+\beta_+ +\beta_- -2}{T.w} \right\rceil + \left\lfloor \tfrac{i+1-\beta_+}{T.w} \right\rfloor - \left\lfloor \tfrac{i+1-\beta_+}{T.w} \right\rfloor + \epsilon_r + \epsilon_d && \text{, } \lceil a+b \rceil \geq \lceil a \rceil + \lfloor b \rfloor \\
&= \left\lceil \tfrac{n+\beta_+ +\beta_- -2}{T.w} \right\rceil + \epsilon_r + \epsilon_d && \text{, simplification} \\
&= \left\lceil \tfrac{n+\beta-2}{T.w} \right\rceil + \epsilon && \text{, by (6) and (7)}
\end{aligned}
$$

This completes the proof. □

Guaranteed allocation bounds. We now derive bounds on the amount of processor time guaranteed to a task under the global scheduler. We begin by proving the following theorem, which bounds the allocation granted to any Pfair task over the interval $[0, t)$.

Theorem 2 *The amount of processor time received by a task T over the interval $[0, t)$, where t is an integer, under scheduling characterized by β_-, β_+, ϵ_r, and ϵ_d, is bounded as shown below.*

$$(\lfloor T.w \cdot (t - \epsilon_d) - \beta_+ \rfloor + 1) \cdot Q \leq \mathit{received}(T, 0, t) \leq (\lceil T.w \cdot (t + \epsilon_r) + \beta_- \rceil - 1) \cdot Q$$

Proof. The proof consists of two parts. First, we restrict attention to the impact of relaxed lag constraints. After addressing this impact, we then consider the impact of ϵ_r and ϵ_d.

Combining (1), (2), and (5) produces

$$-\beta_- \cdot Q < Q \cdot T.w \cdot t - \mathit{received}(T, 0, t) < \beta_+ \cdot Q,$$

which must hold for all values of t. Dividing all terms in the previous inequality by Q and rearranging terms to isolate $\frac{1}{Q} \cdot \mathit{received}(T, 0, t)$ yields the following inequality.

$$T.w \cdot t - \beta_+ < \frac{1}{Q} \cdot \mathit{received}(T, 0, t) < T.w \cdot t + \beta_-$$

By the statement of the theorem, t is an integer, and hence $Q \mid \mathit{received}(T, 0, t)$ (*i.e.*, $\mathit{received}(T, 0, t)$ is divisible by Q) holds since processor time is allocated in units of Q. It follows that $\frac{1}{Q} \cdot \mathit{received}(T, 0, t)$ is an integer. When $\frac{a}{b}$ is an integer, then $x < \frac{a}{b} < y \Leftrightarrow \lfloor x \rfloor + 1 \leq \frac{a}{b} \leq \lceil y \rceil - 1 \Leftrightarrow (\lfloor x \rfloor + 1) \cdot b \leq a \leq (\lceil y \rceil - 1) \cdot b$.

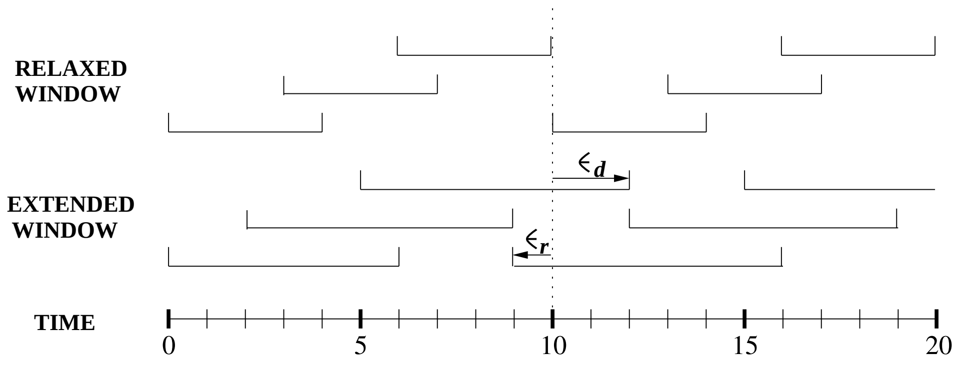


Figure 11: The figure shows the task $\mathbf{PF}\left(\frac{3}{10}\right)$ when $\beta_- = \beta_+ = 1$ (*i.e.*, with Pfair lag bounds). The window layouts show the impact of ϵ_r and ϵ_d.

This property implies that the previous inequality can be rewritten as

$$(\lfloor T.w \cdot t - \beta_+ \rfloor + 1) \cdot Q \leq \mathit{received}(T, 0, t) \leq (\lceil T.w \cdot t + \beta_- \rceil - 1) \cdot Q.$$

Intuitively, $\lfloor T.w \cdot t - \beta_+ \rfloor + 1$ (respectively, $\lceil T.w \cdot t + \beta_- \rceil - 1$) is the number of subtasks with relaxed deadlines at or before (respectively, with relaxed releases before) time t.

We now consider the impact of ϵ_r and ϵ_d on the above allocation bounds. A subtask with a relaxed deadline at time t_d may not complete until its extended deadline at time $t_d + \epsilon_d$. Hence, the lower bound may be smaller than that given above. For instance, consider Figure 11, which shows the relaxed and extended windows for a task $T = \mathbf{PF}\left(\frac{3}{10}\right)$ when $\beta_- = \beta_+ = 1$, $\epsilon_r = 1$, and $\epsilon_d = 2$. The lower bound of *received*$(T, 0, 10)$ with respect to relaxed windows includes T_3 since its relaxed deadline occurs at time 10. However, when considering extended windows, T_3 cannot be counted in the lower bound since it may be executed after time 10. In general, the number of subtasks with extended deadlines at or before time t equals the number with relaxed deadlines at or before time $t - \epsilon_d$. By the lower bound derived above for relaxed windows, $\lfloor T.w \cdot t - \beta_+ \rfloor + 1$ subtasks have relaxed deadlines at or before time t. It follows that the lower bound with respect to extended windows is given by $(\lfloor T.w \cdot (t - \epsilon_d) - \beta_+ \rfloor + 1) \cdot Q$.

Similarly, subtasks with relaxed releases before $t + \epsilon_r$ may be scheduled before t when using extended windows. Hence, the upper bound of *received*$(T, 0, t)$ may be larger than that given above. Again, consider *received*$(T, 0, 10)$ in Figure 11. T_4 has a relaxed release at time 10. Hence, it must be excluded from the upper bound when considering relaxed windows. However, when using extended windows, the window release occurs at time 9. In general, the number of extended windows with releases at or after time t equals the number of relaxed windows with releases at or after time $t + \epsilon_r$. By the upper bound derived above for relaxed windows, $\lceil T.w \cdot t + \beta_- \rceil - 1$ subtasks have relaxed releases at or after time t. It follows that the upper bound with respect to extended windows is $(\lceil T.w \cdot (t + \epsilon_r) + \beta_- \rceil - 1) \cdot Q$. $\square$

The next theorem extends the above result to an arbitrary interval $[t, t + L)$.

Theorem 3 *The amount of processor time received by a task T over the interval $[t, t+L)$, where t and L are integers, under scheduling characterized by β_-, β_+, ϵ_r, and ϵ_d, is bounded as shown below.*

$$(\lfloor T.w \cdot (L - \epsilon) - \beta \rfloor + 1) \cdot Q \leq \mathit{received}(T, t, t+L) \leq (\lceil T.w \cdot (L + \epsilon) + \beta \rceil - 1) \cdot Q$$

Proof. Due to the absence of IS delays over the interval $[t, t+L)$, it is sufficient to consider the allocation that is guaranteed to a task that never experiences IS delays. Since t and L are integers, Theorem 2 can be applied to derive the lower bound, as shown below.

$$\begin{aligned}
&\mathit{received}(T, t, t+L) \\
&\quad = \mathit{received}(T, 0, t+L) - \mathit{received}(T, 0, t) \\
&\qquad\qquad\qquad\qquad , \text{ by definition} \\
&\quad \geq (\lfloor T.w \cdot ((t+L) - \epsilon_d) - \beta_+ \rfloor + 1) \cdot Q - (\lceil T.w \cdot (t + \epsilon_r) + \beta_- \rceil - 1) \cdot Q \\
&\qquad\qquad\qquad\qquad , \text{ by Theorem 2} \\
&\quad = \lfloor (T.w \cdot (L - \epsilon_d - \epsilon_r) - \beta_+ - \beta_-) + (T.w \cdot (t + \epsilon_r) + \beta_-) \rfloor \cdot Q \\
&\qquad\quad - \lceil T.w \cdot (t + \epsilon_r) + \beta_- \rceil \cdot Q + 2 \cdot Q \\
&\qquad\qquad\qquad\qquad , \text{ by rewriting} \\
&\quad \geq \lceil T.w \cdot (t + \epsilon_r) + \beta_- \rceil \cdot Q + (\lfloor T.w \cdot (L - \epsilon_d - \epsilon_r) - \beta_+ - \beta_- \rfloor - 1) \cdot Q \\
&\qquad\quad - \lceil T.w \cdot (t + \epsilon_r) + \beta_- \rceil \cdot Q + 2 \cdot Q \\
&\qquad\qquad\qquad\qquad , \ \lfloor a + b \rfloor \geq \lceil a \rceil + \lfloor b \rfloor - 1 \\
&\quad = (\lfloor T.w \cdot (L - \epsilon_d - \epsilon_r) - \beta_+ - \beta_- \rfloor + 1) \cdot Q \\
&\qquad\qquad\qquad\qquad , \text{ simplification} \\
&\quad = (\lfloor T.w \cdot (L - \epsilon) - \beta \rfloor + 1) \cdot Q \\
&\qquad\qquad\qquad\qquad , \text{ by definition of } \epsilon \text{ and } \beta
\end{aligned}$$

Similarly, the derivation given below establishes the upper bound.

$$\begin{aligned}
&\mathit{received}(T, t, t+L) \\
&\quad = \mathit{received}(T, 0, t+L) - \mathit{received}(T, 0, t) \\
&\qquad\qquad\qquad\qquad , \text{ by definition} \\
&\quad \leq (\lceil T.w \cdot ((t+L) + \epsilon_r) + \beta_- \rceil - 1) \cdot Q - (\lfloor T.w \cdot (t - \epsilon_d) - \beta_+ \rfloor + 1) \cdot Q \\
&\qquad\qquad\qquad\qquad , \text{ by Theorem 2} \\
&\quad = \lceil (T.w \cdot (L + \epsilon_d + \epsilon_r) + \beta_+ + \beta_-) + (T.w \cdot (t - \epsilon_d) - \beta_+) \rceil \cdot Q \\
&\qquad\quad - \lfloor T.w \cdot (t - \epsilon_d) - \beta_+ \rfloor \cdot Q - 2 \cdot Q \\
&\qquad\qquad\qquad\qquad , \text{ by rewriting} \\
&\quad \leq (\lceil T.w \cdot (L + \epsilon_d + \epsilon_r) + \beta_+ + \beta_- \rceil + 1) \cdot Q + \lfloor T.w \cdot (t - \epsilon_d) - \beta_+ \rfloor \cdot Q \\
&\qquad\quad - \lfloor T.w \cdot (t - \epsilon_d) - \beta_+ \rfloor \cdot Q - 2 \cdot Q \\
&\qquad\qquad\qquad\qquad , \ \lceil a + b \rceil \leq \lceil a \rceil + \lfloor b \rfloor + 1 \\
&\quad = (\lceil T.w \cdot (L + \epsilon_d + \epsilon_r) + \beta_+ + \beta_- \rceil - 1) \cdot Q
\end{aligned}$$

$$, \text{ simplification}$$

$$= (\lceil T.w \cdot (L + \epsilon) + \beta \rceil - 1) \cdot Q$$

$$, \text{ by definition of } \epsilon \text{ and } \beta$$

The combination of these bounds establishes the theorem. □