Real-Time Scheduling on Multicore Platforms (Full Version) [*]

James H. Anderson, John M. Calandrino, and UmaMaheswari C. Devi

Department of Computer Science

The University of North Carolina at Chapel Hill

October 2005

Abstract

Multicore architectures, which have multiple processing units on a single chip, are widely viewed as a way to achieve higher processor performance, given that thermal and power problems impose limits on the performance of single-core designs. Accordingly, several chip manufacturers have already released, or will soon release, chips with dual cores, and it is predicted that chips with up to 32 cores will be available within a decade. To effectively use the available processing resources on multicore platforms, software designs should avoid co-executing applications or threads that can worsen the performance of shared caches, if not thrash them. While cache-aware scheduling techniques for such platforms have been proposed for throughput-oriented applications, to the best of our knowledge, no such work has targeted real-time applications. In this paper, we propose and evaluate a cache-aware Pfair-based scheduling scheme for real-time tasks on multicore platforms.

Keywords: Multicore architectures, multiprocessors, real-time scheduling.

[*] Work supported by NSF grants CCR 0309825 and CNS 0408996. The third author was also supported by an IBM Ph.D. fellowship.

1 Introduction

Thermal and power problems limit the performance that single-processor chips can deliver. Multicore architectures, or chip multiprocessors, which include several processors on a single chip, are being widely touted as a solution to this problem. Several chip makers have released, or will soon release, dual-core chips. Such chips include Intel's Pentium D and Pentium Extreme Edition, IBM's PowerPC, AMD's Opteron, and Sun's UltraSPARC IV. A few designs with more than two cores have also been announced. For instance, Sun expects to ship its eight-core Niagara chip by early 2006, while Intel is expected to release four-, eight-, 16-, and perhaps even 32-core chips within a decade [20].

In many proposed multicore platforms, different cores share either on- or off-chip caches. To effectively exploit the available parallelism on these platforms, shared caches must not become performance bottlenecks. In this paper, we consider this issue in the context of real-time applications. To reasonably constrain the discussion, we henceforth limit attention to the multicore architecture shown in Fig. 1, wherein all cores are symmetric and share a chip-wide L2 cache. This general architecture has been widely studied. Of greatest relevance to this paper is prior work by Fedorova *et al.* [12] pertaining to throughput-oriented systems. They noted that L2 misses affect performance to a much greater extent than L1 misses. This is because the cost of an L2 miss can be as high as 100-300 cycles, while the penalty of an L1 miss that can be serviced by the L2 cache is just a few cycles. Based on this fact, Fedorova *et al.* proposed an approach for improving throughput by reducing L2 contention. In this approach, threads that generate significant memory-to-L2 traffic are discouraged from being co-scheduled.

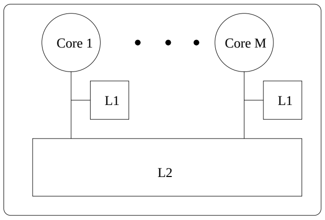


Figure 1: Multicore architecture.

The problem. The problem addressed herein is motivated by the work of Fedorova *et al.*—we wish to know whether, in real-time systems, tasks that generate significant memory-to-L2 traffic can be discouraged from being co-scheduled *while ensuring real-time constraints*. Our focus on such constraints (instead of throughput) distinguishes our work from Fedorova *et al.*'s. In addition, for simplicity, we assume that each core supports one hardware thread, while they considered multithreaded systems.

Other related work. The only other related paper on multicore systems known to us is one by Kim *et al.* [15], which is also directed at throughput-oriented applications. In this paper, a cache-partitioning scheme is presented that uniformly distributes the impact of cache contention among co-scheduled threads.

In work on (non-multicore) systems that support *simultaneous multithreading* (*SMT*), prior work on *symbiotic scheduling* is of relevance to our work [14, 18, 21]. In symbiotic scheduling, the goal is to maximize the overall "symbiosis factor," which is a measure that indicates how well various thread groupings perform when co-scheduled. To the best of our knowledge, no analytical results concerning real-time constraints have been obtained in work on symbiotic scheduling.

Proposed approach. The need to discourage certain tasks from being co-scheduled fundamentally distinguishes the problem at hand from other real-time multiprocessor scheduling problems considered previously [8]. Our approach for doing this is a two-step process: **(i)** combine tasks that may induce significant memory-to-L2 traffic into groups; **(ii)** at runtime, use a scheduling policy that reduces concurrency within groups.

The group-cognizant scheduling policy we propose is a hierarchical scheduling approach based on the concept of a *megatask*. A megatask represents a task group and is treated as a single schedulable entity. A top-level scheduler allocates one or more processors to a megatask, which in turn allocates them to its component tasks. Let γ be a megatask comprised of component tasks with total utilization $I + f$, where I is integral and $0 < f < 1$. (If $f = 0$, then component-task scheduling is straightforward.) Then, the component tasks of γ require between I and $I + 1$ processors for their deadlines to be met. This means that it is impossible to guarantee that fewer than I of the tasks in γ execute at any time. If co-scheduling this many tasks in γ can thrash the L2 cache, then the system simply must be re-designed. In this paper, we propose a scheme that ensures that at most $I + 1$ tasks in γ are ever co-scheduled, which is the best that can be hoped for.

Example. Consider the following four-core example in which the objective is to ensure that the combined working-set size [11] of the tasks that are co-scheduled does not exceed the capacity of the L2 cache. Let the task set τ be comprised of three tasks each of weight (*i.e.*, utilization) 0.6 and working-set size 200 KB (Group A), and four tasks each of weight 0.3 and working-set size 50 KB (Group B). (The weights of the tasks are assumed to be in the absence of heavy L2 contention.) Let the capacity of the L2 cache be 512 KB. The total weight of τ is 3, so co-scheduling at least three of its tasks is unavoidable. However, since the combined working-set size of the tasks in Group A exceeds the L2 capacity, it is desirable that the three co-scheduled tasks not all be from this group. Because the total utilization of Group A is 1.8, by combining the tasks in Group A into a single megatask, it can be ensured that at most two tasks from it are ever co-scheduled.

Contributions. Our contributions in this paper are four-fold. First, we propose a scheme for incorporating megatasks into a Pfair-scheduled system. Our choice of Pfair scheduling is due to the fact that it is the only known way of optimally scheduling recurrent real-time tasks on multiprocessors [5, 22]. This optimality is achieved at the expense of potentially frequent task migrations. However, *multicore architectures tend to mitigate this weakness, as long as L2 miss rates are kept low*. This is because, in the absence of L2 misses, migrations merely result in L1 misses, which do not constitute a significant expense. Second, we show that if a megatask is scheduled using its *ideal* weight (*i.e.*, the cumulative weight of its component tasks), then its component tasks may miss their deadlines, but such misses can be avoided by slightly inflating the megatask's weight. Third, we show that if a megatask's weight is not increased, then component-task deadlines are missed by a bounded amount only, which may be sufficient for soft real-time systems. Finally, through extensive experiments on a multicore simulator, we evaluate the improvement in L2 cache behavior that our scheme achieves in comparison to both a cache-oblivious Pfair scheduler and a partitioning-based scheme. In these experiments, the use of megatasks resulted in significant L2 miss-rate reductions (a reduction from 90% to 2% occurred in one case—see Table 2 in Sec. 4). Indeed, megatask-based Pfair scheduling proved to be the superior scheme from a performance standpoint, and its use was much more likely to result in a schedulable system in comparison to partitioning.

In the rest of the paper, we present an overview of Pfair scheduling (Sec. 2), discuss megatasks and their properties (Sec. 3), present our experimental evaluation (Sec. 4), and discuss avenues for further work (Sec. 5).

2 Background on Pfair Scheduling

Pfair scheduling [5, 22] can be used to schedule a *periodic*, *intra-sporadic* (*IS*), or *generalized-intra-sporadic* (*GIS*) (see below) task system τ on $M \geq 1$ processors. Each task T of τ is assigned a rational weight $wt(T) \in (0, 1]$ that denotes the processor share it requires. For a periodic task T, $wt(T) = T.e/T.p$, where $T.e$ and $T.p$ are the (integral) *execution cost* and *period* of T. A task is *light* if its weight is less than $1/2$, and *heavy*, otherwise.

Pfair algorithms allocate processor time in discrete quanta; the time interval $[t, t+1)$, where $t \in \mathbb{N}$ (the set of nonnegative integers), is called *slot* t. (Hence, time t refers to the beginning of slot t.) All references to time are non-negative integers. Hence, the interval $[t_1, t_2)$ is comprised of slots t_1 through $t_2 - 1$. A task may be allocated time on different processors, but not in the same slot (*i.e.*, interprocessor migration is allowed but parallelism is not). A Pfair schedule is formally defined by a function $\mathcal{S} : \tau \times \mathbb{N} \mapsto \{0, 1\}$, where $\sum_{T \in \tau} \mathcal{S}(T, t) \leq M$ holds for all t. $\mathcal{S}(T, t) = 1$ iff T is scheduled in slot t.

Periodic and IS task models. In Pfair scheduling, each task T is divided into an infinite sequence of quantum-length *subtasks*, $T_1, T_2, \cdots$. Each subtask T_i has an associated *release* $r(T_i)$ and *deadline* $d(T_i)$, defined as follows.

$$r(T_i) = \theta(T_i) + \left\lfloor \frac{i-1}{wt(T)} \right\rfloor \ \wedge \ d(T_i) = \theta(T_i) + \left\lceil \frac{i}{wt(T)} \right\rceil \tag{1}$$

In (1), $\theta(T_i)$ denotes the *offset* of T_i. The offsets of T's various subtasks are nonnegative and satisfy the following:

$$k > i \Rightarrow \theta(T_k) \geq \theta(T_i). \tag{2}$$

T is *periodic* if $\theta(T_i) = c$ holds for all i (and is *synchronous* also if $c = 0$), and is *IS*, otherwise. Examples are given in insets (a) and (b) of Fig. 2. The restriction on offsets implies that the separation between any pair of subtask releases is at least the separation between those releases if the task were periodic. The interval $[r(T_i), d(T_i))$ is termed the *window* of T_i. The lemma below concerning window lengths follows from (1).

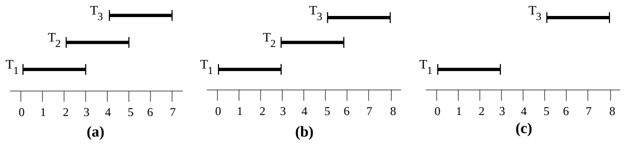


Figure 2: **(a)** Windows of the first three subtasks of a periodic task T with weight $3/7$. **(b)** Windows of an IS task. Subtask T_2 is released one time unit late. **(c)** Windows of a GIS task. T_2 is absent and T_3 is released one time unit late.

Lemma 1 (from [4]) *The length of any window of a task T is either $\left\lceil \frac{1}{wt(T)} \right\rceil$ or $\left\lceil \frac{1}{wt(T)} \right\rceil + 1$.*

GIS task model. A GIS task system is obtained by removing subtasks from a corresponding IS (or GIS) task system. Specifically, in a GIS task system, a task T, after releasing subtask T_i, may release subtask T_k, where $k > i + 1$, instead of T_{i+1}, with the following restriction: $r(T_k) - r(T_i)$ is at least $\left\lfloor \frac{k-1}{wt(T)} \right\rfloor - \left\lfloor \frac{i-1}{wt(T)} \right\rfloor$. In other words, $r(T_k)$ is not smaller than what it would have been if $T_{i+1}, T_{i+2}, \ldots, T_{k-1}$ were present and released as early as possible. For the special case where T_k is the first subtask released by T, $r(T_k)$ must be at least $\left\lfloor \frac{k-1}{wt(T)} \right\rfloor$. Fig. 2(c) shows an example. Note that a periodic task system is an IS task system, which in turn is a GIS task system, so any property established for the GIS task model applies to the other models, as well.

Pfair scheduling algorithms. Pfair scheduling algorithms schedule tasks by choosing at most M eligible subtasks at the beginning of every time slot. At present, three optimal Pfair scheduling algorithms —PF [5], PD [6], and PD^2 [4, 22]—and one suboptimal algorithm—earliest pseudo-deadline first (EPDF) [4]—are known. An *optimal* algorithm correctly schedules any GIS task system τ for which $\sum_{T \in \tau} wt(T) \leq M$ holds. In all of these algorithms, a subtask with an earlier deadline has a higher priority than one with a later deadline. The optimal algorithms use additional rules to resolve ties among subtasks with the same deadline. In fact, the three optimal algorithms differ only in their tie-breaking rules; PD^2 is the most efficient of the three and its tie-breaking rules subsume those of the other two algorithms. The suboptimal EPDF algorithm uses no tie-breaking rules, but resolves all ties arbitrarily.

3 Megatasks

A megatask is simply a set of *component* tasks to be treated as a single schedulable entity. The notion of a megatask extends that of a supertask, which was proposed in previous work [17]. In particular, the cumulative weight of a megatask's component tasks may exceed one, while a supertask may have a total weight of at most one. For simplicity, we will henceforth call such a task grouping a *megatask* only if its cumulative weight *exceeds* one; otherwise, we will call it a *supertask.* A task system τ may consist of $g \geq 0$ megatasks, with the j^{th} megatask denoted γ^j. Tasks in τ are independent and each task may be included in at most one megatask. A task that is not included in any megatask is said to be *free.* (Some of these free tasks may in fact be supertasks, but this is not a concern for us.) The cumulative weight of the component tasks of γ^j, denoted $W_{sum}(\gamma^j)$, can be expressed as $I_j + f_j$, where I_j is a positive integer and $0 \leq f_j < 1$. $W_{sum}(\gamma^j)$ is also referred to as the *ideal weight* of γ^j. We let $W_{\max}(\gamma^j)$ denote the maximum weight of any component task of γ^j. (To reduce clutter, we often omit both the j superscripts and subscripts and also the megatask γ^j in W_{sum} and $W_{\max}$.)

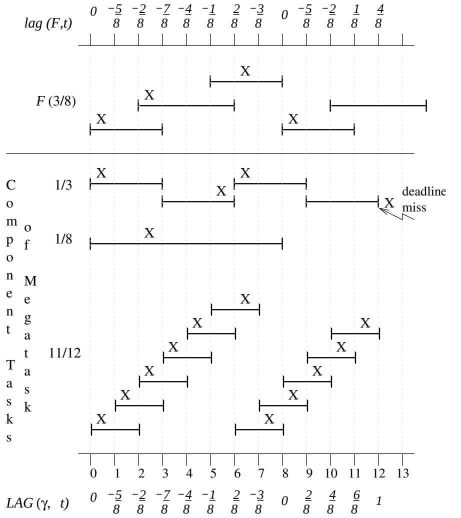


Figure 3: PD^2 schedule for the component (GIS) tasks of a megatask γ with $W_{sum} = 1 + \frac{3}{8}$. F represents the fictitious task associated with γ. γ is scheduled using its ideal weight by a top-level PD^2 scheduler. The slot in which a subtask is scheduled is indicated using an "X." γ is allocated two processors in slots where F is scheduled and one processor in the remaining slots. In this schedule, one of the processors allocated to γ at time 8 is idled and a deadline is missed at time 12.

The megatask-based scheduling scheme we propose is a two-level hierarchical approach. The root-level scheduler is PD^2, which schedules all megatasks and free tasks of τ. Pfair scheduling with megatasks is a straightforward extension to ordinary Pfair scheduling wherein a dummy or fictitious, synchronous, periodic task F^j of weight f_j is associated with megatask γ^j, I_j processors are statically assigned to γ^j in every slot, and $M - \sum_{\ell=1}^{g} I_\ell$ processors are allocated at runtime to the fictitious tasks and free tasks by the root-level PD^2 scheduler. Whenever task F^j is scheduled, an additional processor is allocated to γ^j.

Unfortunately, even with the optimal PD^2 algorithm as the second-level scheduler, component-task deadlines may be missed. Fig. 3 shows a simple example. Hence, the principal question that we address in this paper is the following: *With two-level hierarchical scheduling as described above, what weight should be assigned to a megatask to ensure that its component-task* *deadlines are met?* We refer to this inflated weight of a megatask as its *scheduling weight*, denoted W_{sch}. Holman and Anderson answered this question for supertasks [13]. However, megatasks require different reasoning. In particular, uniprocessor analysis techniques are sufficient for supertasks (since they have total weight at most one), but not megatasks. In addition, unlike a supertask, the fractional part (f) of a megatask's ideal weight may be less than $W_{\max}$. Hence, there is not much semblance between the approach used in this paper and that in [13].

Other applications of megatasks. Megatasks may also be used in systems wherein disjoint subsets of tasks are constrained to be scheduled on different subsets of processors. The existence of a Pfair schedule for a task system with such constraints is proved in [16]. However, no optimal or suboptimal online Pfair scheduling algorithm has been proposed so far for this problem.

In addition, megatasks can be used to schedule tasks that access common resources as a group. Because megatasks restrict concurrency, their use may enable the use of less expensive synchronization techniques and result in less pessimism when determining synchronization overheads (*e.g.*, blocking times).

Megatasks might also prove useful in providing an *open-systems* [10] infrastructure that temporally isolates independently-developed applications running on a common platform. In work on open systems, a two-level scheduling hierarchy is usually used, where each node at the second level corresponds to a different application. All prior work on open systems has focused only on uniprocessor platforms or applications that require the processing capacity of at most one processor. A megatask can be viewed as a multiprocessor server and is an obvious building block for extending the open-systems architecture to encompass applications that exceed the capacity of a single processor.

Comparison of megatasks and supertasks for cache-aware scheduling. Before proceeding further, we address the question of whether a megatask can be replaced by a group of supertasks for the cache-contention problem, and compare the overheads in terms of weight inflation that the two approaches entail. First, we show that there exist problem instances that cannot be solved using supertasking, but by megatasking. For this, consider the following example involving six tasks with weights $\frac{5}{6}$, $\frac{5}{6}$, $\frac{1}{5}$, $\frac{1}{5}$, $\frac{4}{15}$, and $\frac{11}{30}$, at most three of which can be scheduled at a time. It can be verified that this task set cannot be partitioned into less than four subsets of cumulative weight at most one. Hence, with supertasking, four supertasks will be required and up to that many tasks may be co-scheduled. On the other hand, since the cumulative weight of the entire task set is only $2\frac{7}{10}$, at most three tasks would be co-scheduled if the tasks are packed into a single megatask. It is easy to see that any megatask γ would have to be decomposed into at least $\lceil W_{sum} \rceil$ supertasks, and hence, if a problem instance that restricts the number of tasks that can be co-scheduled is feasible using supertasking, then it is feasible using megatasking, as well.

We next show that for *feasible* problem instances, megatasking and supertasking are incomparable, *i.e.*, there are problem instances for which the weight inflation required is lower with megatasking than with supertasking, and vice versa. For this, first consider three tasks with weights $\frac{3}{8}$, $\frac{1}{3}$, and $\frac{1}{3}$. The cumulative weight of these three tasks is $1\frac{1}{24}$. If these tasks are packed into a megatask, then, using the rules in the next subsection, it is sufficient to increase the scheduling weight of the megatask by $\frac{1}{24}$ to $1\frac{1}{12}$. On the other hand, if we were to use supertasking, then at least two supertasks will be required. Even with the better of the two possible packings, wherein the first supertask contains the first two component tasks, and the second contains just the third task, using the rules of [13], the weight of the first supertask will have to be increased by $\frac{7}{24}$, which is $\frac{1}{4}$ higher than $\frac{1}{24}$. Thus, in this case supertasking entails more overhead. For an example in which megatasking is costlier than supertasking, consider the task set in Fig. 3. The scheduling weight of the megatask here will have to be increased to $1\frac{9}{11}$, which is $\frac{39}{88}$ higher than its ideal weight, whereas, if the tasks are packed into two supertasks, with the task with weight $\frac{11}{12}$ in a supertask of its own, and the remaining two tasks in a second supertask, then only the second supertask's weight would have to be increased to $\frac{2}{3}$. Thus, weight inflation required is greater with the megatask approach by $\frac{31}{132}$. Though in general the two approaches are incomparable, we can expect megatasking to perform better on an average. This is because, if the cumulative weight of all component tasks exceeds two, then at least three supertasks would be required to pack them, and more than one supertask may require a weight inflation, and we can expect the sum of multiple weight inflations to be higher than a single weight inflation required for a megatask. Apart from lower weight inflations, conceptually megatasking is simpler and is a more suitable abstraction for the problem under consideration than supertasking.

Reweighting a megatask. We now present reweighting rules that can be used to compute a megatask scheduling weight that is sufficient to avoid deadline misses by its component tasks when PD^2 is used as both the top- and second-level scheduler. Let W_{sum}, $W_{\max}$, and $\omega_{\max}$ be defined as follows. ($\omega_{\max}$ denotes the smaller of the at most two window lengths of a task with weight $W_{\max}$—refer to Lemma 1.)

$$\begin{aligned} W_{sum} &= \textstyle\sum_{T\in\gamma} wt(T) = I + f && (3) \\ W_{\max} &= \max_{T\in\gamma} wt(T) && (4) \\ \omega_{\max} &= \lceil 1/W_{\max} \rceil && (5) \end{aligned}$$

Let the *rank of a component task* of γ be its position in a non-increasing ordering of the component tasks by weight. Let ω be as follows. (In this paper, $W_{\max} = \frac{1}{k}$ is used to denote that $W_{\max}$ can be expressed as the reciprocal of an arbitrary positive integer.)

$$\omega = \begin{cases} \min(\text{smallest window length of task of rank } (\omega_{\max} \cdot I + 1), 2\omega_{\max}), & \text{if } W_{\max} = \frac{1}{k},\ k \in \mathbb{N}^+ \\ \min(\text{smallest window length of task of rank } ((\omega_{\max} - 1) \cdot I + 1), 2\omega_{\max} - 1), & \text{otherwise} \end{cases} \quad (6)$$

Then, a scheduling weight W_{sch} for γ may be computed using (7), where Δ_f is given by (8).

$$W_{sch} = W_{sum} + \Delta_f \quad (7)$$

$$\Delta_f = \begin{cases} \left(\frac{W_{\max} - f}{1 + f - W_{\max}}\right) \times f, & \text{if } W_{\max} \geq f + 1/2 \\ \min(1 - f, \max(\left(\frac{W_{\max} - f}{1 + f - W_{\max}}\right) \times f, \min(f, \frac{1}{\omega - 1}))), & \text{if } f + 1/2 > W_{\max} > f \\ \min(1 - f, \frac{1}{\omega}), & \text{if } W_{\max} \leq f \\ 0, & \text{if } f = 0 \end{cases} \quad (8)$$

Reweighting example. Let γ be a megatask with two component tasks of weight $\frac{2}{5}$ each, and three more tasks of weight $\frac{1}{4}$ each. Hence, $W_{\max} = \frac{2}{5}$ and $W_{sum} = I + f = 1\frac{11}{20}$, so, $I = 1$, $f = \frac{11}{20}$. Since $W_{\max} < f$, by (8), $\Delta_f = \min(1 - f, \frac{1}{\omega})$. We determine ω as follows. By (5), $\omega_{\max} = 3$. Since $W_{\max} \neq \frac{1}{k}$, $\omega = \min(\text{smallest window length of task of rank } ((\omega_{\max} - 1) \cdot I + 1), 2\omega_{\max} - 1)$.

$(\omega_{\max} - 1) \cdot I + 1 = 3$, and the weight of the task of rank 3 is $\frac{1}{4}$. By Lemma 1, the smallest window length of a task with weight $\frac{1}{4}$ is 4. Hence, $\omega = \min(4, 5) = 4$, and $\Delta_f = \min(\frac{9}{20}, \frac{1}{4}) = \frac{1}{4}$. Thus, $W_{sch} = W_{sum} + \Delta_f = 1\frac{16}{20}$. ■

Correctness proof. In an appendix, we prove that W_{sch}, given by (7), is a sufficient scheduling weight for γ to ensure that all of its component-task deadlines are met. The proof is by contradiction: we assume that some time t_d exists that is the earliest time at which a deadline is missed. We then determine a bound on the allocations to the megatask up to time t_d and show that, with its weight as defined by (7), the megatask receives sufficient processing time to avoid the miss. This setup is similar to that used by Srinivasan and Anderson in the optimality proof of PD^2 [22]. However, a new twist here is the fact that the number of processors allocated to the megatask is not constant (it is allocated an "extra" processor in some slots). To deal with this issue, some new machinery for the proof had to be devised. From this proof, the theorem below follows.

Theorem 1 *Under the proposed two-level* PD^2 *scheduling scheme, if the scheduling weight of a megatask* γ *is determined by* (7)*, then no component tasks of* γ *miss deadlines.*

Why does reweighting work? In the absence of reweighting, missed component-task deadlines are *not* the result of the megatask being allocated too little processor time. After all, the megatask's total weight in this case matches the combined weight of its component tasks. Instead, such misses result because of mismatches with respect to the *times* at which allocations to the megatask occur. More specifically, misses happen when the allocations to the fictitious task F are "wasted," as seen in Fig. 3.

Reweighting works because, by increasing F's weight, the allocations of the extra processor can be made to align sufficiently with the processor needs of the component tasks so that misses are avoided. In order to minimize the number of wasted processor allocations, it is desirable to make the reweighting term as small as possible. The trivial solution of setting the reweighting term to $1 - f$ (essentially providing an extra processor in all slots), while simple, is incredibly wasteful. The various cases in (8) follow from systematically examining (in the proof) all possible alignments of component-task windows and windows of F.

Tardiness bounds without reweighting. It is possible to show that if a megatask is not reweighted, then its component tasks may miss their deadlines by only a bounded amount. (Note that, when a subtask of a task misses its deadline, the release of its next subtask is not delayed. Thus, if deadline tardiness is bounded, then each task receives its required processor share in the long term.) Due to space constraints, it is not feasible to give a proof of this fact here, so we merely summarize the result. For $W_{\max} \leq f$ (resp., $W_{\max} > f$), if $W_{\max} \leq \frac{I+q-1}{I+q}$ (resp., $W_{\max} \leq \frac{I+q-2}{I+q-1}$) holds, then no deadline is missed by more than q quanta, for all $I \geq 1$ (resp., $I \geq 2$). For $I = 1$ and $W_{\max} > f$, no deadline is missed by more than q quanta, if the weight of every component task is at most $\frac{q-1}{q+1}$. Note that as I increases, the restriction on $W_{\max}$ for a given tardiness bound becomes more liberal.

Aside: determining execution costs. In the periodic task model, task weights depend on per-job execution costs, which depend on cache behavior. In soft real-time systems, profiling tools used in work on throughput-oriented applications [1, 7] might prove useful in determining such behavior. In test applications considered by Fedorova *et al.* [12], these tools proved to be quite accurate, typically producing miss-rate predictions within a few percent of observed values. In hard real-time systems, determining execution costs is a *difficult* timing analysis problem. This problem is made no harder by the use of megatasks—indeed, cache behavior will depend on co-scheduling choices, and with megatasks, more definitive statements regarding such choices can be made. Since multicore systems are likely to become the “standard” platform in many settings, these timing analysis issues are important for the research community to address (and are well beyond the scope of this paper).

4 Experimental Results

To assess the efficacy of megatasking in reducing cache contention, we conducted experiments using the SESC Simulator [19], which is capable of simulating a variety of multicore architectures. We chose to use a simulator so that we could experiment with systems with more cores than commonly available today. The simulated architecture we considered consists of a variable number of cores, each with dedicated 16K L1 data and instruction caches (4- and 2-way set associative, respectively) with random and LRU replacement policies, respectively, and a shared 8-way set associative 512K on-chip L2 cache with an LRU replacement policy. Each cache has a 64-byte line size. Each scheduled task was assigned a utilization and memory block with a given working-set size (WSS). A task accesses its memory block sequentially, looping back to the beginning of the block when the end is reached. We note that all scheduling, preemption, and migration costs were accounted for in these simulations.

The following subsections describe two sets of experiments, one involving hand-crafted example task sets, and a second involving randomly generated task sets. In both sets, Pfair scheduling with megatasks was compared to both partitioned EDF and ordinary Pfair scheduling (without megatasks).

4.1 Hand-Crafted Task Sets

The hand-crafted task sets we created are listed in Table 1. Each was run on either a four- or eight-core machine, as specified, for the indicated number of quanta (assuming a 1-ms quantum length). Table 2 shows for each case the L2 cache-miss rates that were observed (first line of each entry) and the minimum, average, and maximum number of per-task memory accesses completed (second line). In obtaining these results, megatasks were not reweighted because we were more concerned here with cache behavior than timing properties. Reweighting impact was assessed in the experiments described in Sec. 4.2. We begin our discussion by considering the miss-rate results for each task set.

BASIC consists of three heavy-weight tasks. Running any two of these tasks concurrently will not thrash the L2 cache, but running all three will. The total utilization of all three tasks is less than two, but the number of cores is four. Both Pfair and partitioning use more than two cores, causing thrashing. By combining all three tasks into one megatask, thrashing is eliminated. In fact, the difference here is quite dramatic. SMALL_BASIC is a variant of BASIC with tasks of smaller utilization. The results here are similar, but not quite as dramatic.

	No.		**No.**	**No.**
Name	**Tasks**	**Task Properties**	**Cores**	**Quanta**
BASIC	3	Wt. 3/5, WSS 250K	4	100
SMALL_BASIC	5	Wt. 7/20, WSS 250K	4	60
ONE_MEGA	5	Wt. 7/10, WSS 120K	8	50
TWO_MEGA	6	3 with Wt. 3/5, WSS 190K 3 with Wt. 3/5, WSS 60K	8	50

Table 1: Properties of example task sets.

ONE_MEGA and TWO_MEGA give cases where one megatask is better than two and vice versa. In the first case, one megatask is better because using two megatasks of weight 2.1 and 1.4 allows an extra task to run in some quanta. In the second case, using two megatasks ensures that at most two of the 190K-WSS tasks and two of the 60K-WSS tasks run concurrently, thus guaranteeing that their combined WSS is under 512K. Packing all tasks into one megatask ensures that at most four of the tasks run concurrently. However, it does not allow us to specify *which* four. Thus, all three tasks with a 190K WSS could be scheduled concurrently, which is undesirable. Interestingly, placing just these three tasks into a single megatask results in little improvement.

Name	**Partitioning**	**Pfair**	**Megatasks**
BASIC	89.12% (1.73, 1.73, 1.73)	90.35% (1.71, 1.72, 1.72)	2.20% (10.9, 11.1, 11.3)
SMALL_BASIC	17.24% (0.61, 2.01, 4.12)	28.84% (0.48, 1.21, 4.14)	2.89% (3.72, 3.74, 3.77)
ONE_MEGA (1 megatask)	11.07% (1.40, 4.89, 7.27)	11.36% (1.35, 4.83, 7.26)	0.82% (7.06, 7.10, 7.15)
ONE_MEGA (2 megatasks, Wt. 2.1 and 1.4)	11.07% (1.40, 4.89, 7.27)	11.36% (1.35, 4.83, 7.26)	1.79% (6.36, 6.84, 7.20)
TWO_MEGA (1 megatask, all task incl.)	10.94% (0.85, 3.58, 6.32)	10.97% (0.86, 3.59, 6.32)	5.67% (2.55, 4.98, 6.25)
TWO_MEGA (1 megatask, only 190K WSS tasks)	10.94% (0.85, 3.58, 6.32)	10.97% (0.86, 3.59, 6.32)	5.52% (2.56, 5.07, 6.22)
TWO_MEGA (2 megatasks, one each for 190K and 60K tasks)	10.94% (0.85, 3.58, 6.32)	10.97% (0.86, 3.59, 6.32)	1.02% (5.43, 5.85, 6.20)

Table 2: L2 cache miss ratios per task set and (Min., Avg., Max.) per-task memory accesses completed, in millions, for example task sets.

The average memory-access figures given in Table 2 show that megatasking results in substantially better performance. This is particularly interesting in comparing against partitioning, because the better comparable performance of megatasking results despite higher scheduling, preemption, and migration costs. Under partitioning and Pfair, substantial differences were often observed for different tasks in the same task set, even though these tasks have the same weight, and for four of the sets, the same WSS. For example, the number of memory accesses (in millions) for the tasks in SMALL_BASIC was {0.614, 4.123, 0.613, 4.103, 0.613} under partitioning, but {3.755, 3.765, 3.743, 3.717, 3.723} for megatasking. Such nonuniform results led to partitioning having higher *maximum* memory-access values in some cases.

4.2 Randomly-Generated Task Sets

We begin our discussion of the second set of experiments by describing our methodology for generating task sets.

Task-set generation methodology. In generating task sets at random, we limited attention to a four-core system, and considered total WSSs of 768K, 896K, and 1024K, which correspond to 1.5, 1.75, and 2.0 times the size of the L2 cache. These values were selected after examining a number of test cases. In particular, we noted the potential for significant thrashing at the 1.5 point. We further chose the 1.75 and 2.0 points (somewhat arbitrarily) to get a sense of how all schemes would perform with an even greater potential for thrashing.

The WSS distribution we used was bimodal in that large WSSs (at least 128K) were assigned to those tasks with the largest utilizations, and the remaining tasks were assigned a WSS (of at least 1K) from what remained of the combined WSS. We believe that this is a reasonable distribution, as tasks that use more processor time tend to access a larger region of memory. Per-task WSSs were capped at 256K so that at least two tasks could run on the system at any given time. Otherwise, it is unlikely any approach could reduce cache thrashing for these task sets (unless all large-WSS tasks had a combined weight of at most one).

Total system utilizations were allowed to range between 2.0 and 3.5. Total utilizations higher than 3.5 were excluded to give partitioning a better chance of finding a feasible partitioning. Utilizations as low as 2.0 were included to demonstrate the effectiveness of megatasking on a lightly-loaded system. Task utilizations were generated uniformly over a range from some specified minimum to one, exclusive. The minimum task utilization was varied from 1/10 (which makes finding a feasible partitioning easier) to 1/2 (which makes partitioning harder). We generated and ran the same number of task sets for each $\{$task utilization, system utilization$\}$ combination as plotted in Fig. 4, which we discuss later.

In total, 552 task sets were generated. Unfortunately, this does not yield enough samples to obtain meaningful confidence intervals. We were unable to generate more samples because of the length of time it took the simulations to run. The SESC simulator is very accurate, but this comes at the expense of being quite slow. We were only able to generate data for approximately 20 task sets per day running the simulator on one machine. For this reason, longer and more detailed simulations also were not possible.

Justification. Our working set sizes are comparable to those considered by Fedorova *et al.* [12] in their experiments, and our L2 cache size is actually larger than any considered by them. While it is true that proposed systems will have shared caches larger than 512K (*e.g.* the Sun Niagra system mentioned earlier will have at least 3MB), we were somewhat constrained by the slowness of SESC to simulate platforms of moderate size. In addition, it is worth pointing out that WSSs for real-time tasks also have the potential to be much larger. For example, the authors of [9] claim that the WSS for a high-resolution MPEG decoding task, such as that used for HDTV, is about 4.1MB. As another example, statistics presented in [24] show that substantial memory usage is necessary in some video-on-demand applications.

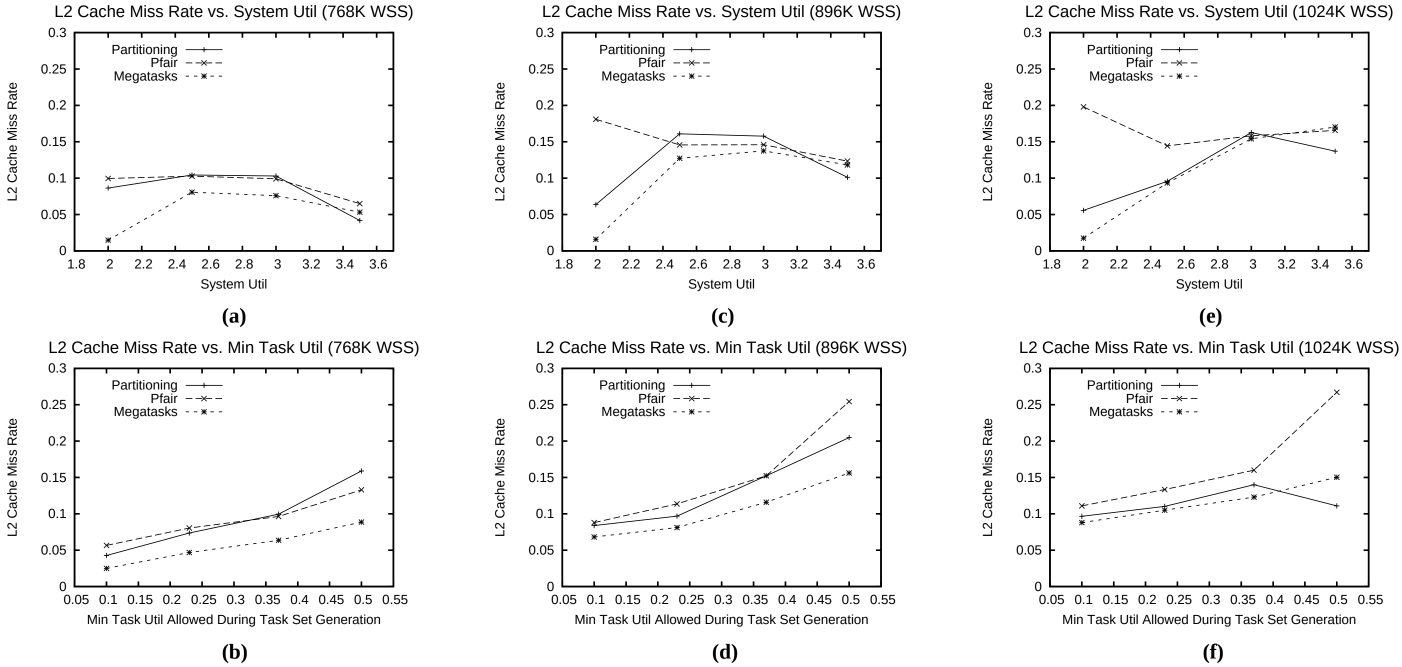
We justify our range of task utilizations, specifically the choice to include heavy tasks, by observing that for a task to access a large region of memory, it typically needs a large amount of processor time. The MPEG decoding application mentioned above is a good example: it requires much more processor time than low-resolution MPEG video decoders. Additionally, our range of task utilizations is similar to that used in other comparable papers [14, 23], wherein tasks with utilizations well-spread among the entire $(0, 1)$ range were considered.

Packing strategies. For partitioning, two attempts to partition tasks among cores were made. First, we placed tasks onto cores in decreasing order of WSS using a first-fit approach. Such a packing, if successful, minimizes the largest possible combined WSS of all tasks running concurrently. If this packing failed, then a second attempt was made by assigning tasks to cores in decreasing order of utilization, again using a first-fit approach. If this failed, then the task set was "disqualified." Such disqualified task sets were not included in the results shown later, but are shown in Table 3.

Algorithm	**No. Disq.**	**% Disq.**
Partitioning	91	16.49
Pfair	0	0.00
Pfair with Megatasks	9	1.63

Table 3: Disqualified task sets for each approach (out of 552 task sets in total).

Tasks were packed into megatasks in order of decreasing WSSs. One megatask was created at a time. If the current task could be added to the current megatask without pushing the megatask's weight beyond the next integer boundary, then this was done, because if the megatask could prevent thrashing among its component tasks before, then it could do so afterwards. Otherwise, a check was made to determine whether creating a new megatask would be better than adding to the current one. While this is an easy packing strategy, it is not necessarily the most efficient. For example, a better packing might be possible by allowing a new task to be added to a megatask generated prior to the current one. For this reason, we believe that the packing strategies we used treat partitioning



(a)

(c)

(e)

(b)

(d)

(f)

Figure 4: L2 cache miss rate versus both total system utilization (top) and minimum task utilization (bottom). The different columns correspond (left to right) to total WSSs of 1.5, 1.75, and 2.0 times the L2 cache capacity, respectively.

more fairly than megatasking.

After creating the megatasks, each was reweighted. If this caused the total utilization to exceed the number of cores, then that task set was "disqualified" as with partitioning. As Table 3 shows, the number of megatask disqualifications was an order of magnitude less than partitioning, even though our task-generation process was designed to make feasible partitionings more likely, and we were using a rather simple megatask packing approach.

Results. Under each tested scheme, each non-disqualified task set was executed for 20 quanta and its L2 miss rates were recorded. Fig. 4 shows the recorded miss rates as a function of the total system utilization (top) and minimum per-task utilization (bottom). The three columns correspond to the three total WSSs tested, *i.e.*, 1.5, 1.75, and 2.0 times the L2 cache size. Each point is an average obtained from between 19 and 48 task sets. (This variation is due to discarded task sets, primarily in the partitioning case, and the way the data is organized.) In interpreting this data, note that, because an L2 miss incurs a time penalty roughly two orders of magnitude greater than a hit, even when miss rates are relatively low, a miss-rate difference can correspond to a significant difference in performance. For example, see Fig. 5, which gives the number of cycles-per-memory-reference for the data shown in Fig. 4(b). Although the speed of the SESC simulator severely constrained the number and length of our simulations, we also ran a small subset of our task sets for 100 quanta (as opposed to 20) and saw approximately the same results. This further justifies 20 quanta as a reasonable "stopping point."

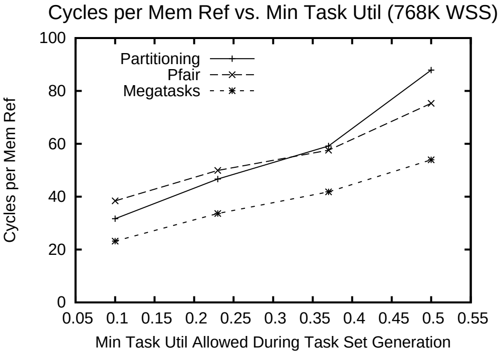


Figure 5: Cycles-per-memory-reference for the data shown in Fig. 4(b).

As seen in the bottom-row plots, the L2 miss rate increases with increasing task utilizations. This is because the heaviest tasks have the largest WSSs and thus are harder to place onto a small number of cores. The top-row plots show a similar trend as the total system utilization increases from 2.0 to 2.5. Beyond this point, however, miss rates level off or decrease. One explanation for this may be that our task-generation process may leave little room to improve miss rates at total utilizations beyond 2.5. The fact that the three schemes approximately converge beyond this point supports this conclusion. With respect to total WSS, at 1.5 times the L2 cache size (left column), megatasking is the clear winner. At 1.75 times (middle column) and 2.0 times (right column), megatasking is still the winner in most cases, but less substantially, because all schemes are less able to improve L2 cache performance. This is particularly noticeable in the 2.0-times case.

Two anomalies are worth noting. First, in inset (e), Pfair slightly outperforms megatasking at the 3.5 system-utilization point. This may be due to miss-rate differences in the scheduling code itself. Second, at the right end point of each plot (3.5 system utilization or 0.5 task utilization), partitioning sometimes wins over the other two schemes, and sometimes loses. These plots, however, are misleading in that, at high utilizations, many of the task sets were disqualified under partitioning. Thus, the data at these points is somewhat skewed. With only non-disqualified task sets plotted (not shown), all three schemes have similar curves, with megatasking always winning.

In addition to the data shown, we also performed similar experiments in which per-task utilizations were capped. We found that, as these caps are lowered, the gap between megatasking and partioning narrows, with megatasking always either winning or, at worst, performing nearly identically to partitioning.

As before, we tabulated memory-access statistics, but this time on a per-task-set rather than per-task basis. (For each scheme, only non-disqualified task sets under it were considered.) These results, as well as instruction counts, are given in Table 4. These statistics exclude the scheduling code itself. Thus, these results should give a reasonable indication of how the different migration, preemption, and scheduling costs of the three schemes impact the amount of "useful work" that is completed. As seen, megatasking is the clear winner by 5-6% on average and by as much as 30% in the worst case (as seen by the minimum values).

Algorithm	**No. Instr.**	**No. Mem. Acc.**
Partitioning	(177.36, 467.83, 647.64)	(51.51, 131.50, 182.20)
Pfair	(229.87, 452.41, 613.77)	(65.96, 124.21, 178.04)
Pfair with Megatasks	(232.23, 495.47, 666.62)	(66.16, 137.62, 182.41)

Table 4: (Min., Avg., Max.) instructions and memory accesses completed over all non-disqualified task sets for each scheduling policy, in millions. From Table 3, every (almost every) task set included in the paritioning counts is included in the Pfair (megatasking) counts.

These experiments should certainly not be considered definitive. Indeed, devising a meaningful random task-set generation process is not easy, and this is an issue worthy of further study. Nonetheless, for the task sets we generated, megatasking is clearly the best scheme. Its use is much more likely to result in a schedulable system, in comparison to partitioning, and also in lower L2 miss rates (and as seen in Sec. 4.1, for some specific task sets, miss rates may be *dramatically* less).

5 Concluding Remarks

We have proposed the concept of a megatask as a way to reduce miss rates in shared caches on multicore platforms. We have shown that deadline misses by a megatask's component tasks can be avoided by slightly inflating its weight and by using Pfair scheduling algorithms to schedule all tasks. We have also given deadline tardiness thresholds that apply in the absence of reweighting. Finally, we have assessed the benefits of megatasks through an extensive experimental investigation. While the theoretical superiority of Pfair-related schemes over other approaches is well known, these experiments are the first (known to us) that show a clear performance advantage of such schemes over the most common multiprocessor scheduling approach, partitioning.

Our results suggest a number of avenues for further research. First, more work is needed to determine if the deadline tardiness bounds given in Sec. 3 are tight. Second, we would like to extend our results for SMT systems that support multiple hardware thread contexts per core, as well as asymmetric multicore designs. Third, as noted earlier, timing analysis on multicore systems is a subject that deserves serious attention. Fourth, we have only considered static, independent tasks in this paper. Dynamic task systems and tasks with dependencies warrant attention as well. Fifth, in some systems, it may be useful to actually *encourage* some tasks to be co-scheduled, as in symbiotic scheduling [14, 18, 21]. Thus, it would be interesting to incorporate symbiotic scheduling techniques within megatasking. Finally, a task's weight may actually depend on how tasks are grouped, because its execution rate will depend on cache behavior. This gives rise to an interesting synthesis problem: as task groupings are determined, weight estimates will likely reduce, due to better cache behavior, and this may enable better groupings. Thus, the overall system design process may be iterative in nature.

References

[1] A. Agarwal, M. Horowitz, and J. Hennessy. An analytical cache model. *ACM Trans. on Comp. Sys.*, 7(2):184–215, 1989.

[2] J. Anderson and A. Srinivasan. Early-release fair scheduling. *Proc. of the 12th Euromicro Conf. on Real-Time Sys.*, pp. 35–43, 2000.

[3] J. Anderson and A. Srinivasan. Pfair scheduling: Beyond periodic task systems. *Proc. of the 7th Int'l Conf. on Real-Time Comp. Sys. and Applications*, pp. 297–306, 2000.

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

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

[6] S. Baruah, J. Gehrke, and C. Plaxton. Fast scheduling of periodic tasks on multiple resources. *Proc. of the 9th International Parallel Processing Symposium*, pp. 280–288, Apr. 1995.

[7] E. Berg and E. Hagersten. Statcache: A probabilistic approach to efficient and accurate data locality analysis. *Proc. of the 2004 IEEE Int'l Symp. on Perf. Anal. of Sys. and Software*, 2004.

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

[9] H. Chen, K. Li, and B. Wei. Memory performance optimizations for real-time software HDTV decoding. *Journal of VLSI Signal Processing*, pp. 193–207, 2005.

[10] Z. Deng, J.W.S. Liu, L. Zhang, M. Seri, and A. Frei. An open environment for real-time applications. *Real-Time Sys. Journal*, 16(2/3):155–186, 1999.

[11] P. Denning. Thrashing: Its causes and prevention. *Proc. of the AFIPS 1968 Fall Joint Comp. Conf.*, Vol. 33, pp. 915–922, 1968.

[12] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. *Proc. of the USENIX 2005 Annual Technical Conf.*, 2005. (See also Technical Report TR-17-04, Div. of Engineering and Applied Sciences, Harvard Univ. Aug., 2004.)

[13] P. Holman and J. Anderson. Guaranteeing Pfair supertasks by reweighting. *Proc. of the 22nd Real-Time Sys. Symp.*, pp. 203–212, 2001.

[14] R. Jain, C. Hughs, and S Adve. Soft real-time scheduling on simultaneous multithreaded processors. *Proc. of the 23rd Real-Time Sys. Symp.*, pp. 134–145, 2002.

[15] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning on a chip multiprocessor architecture. *Proc. of the Parallel Architecture and Compilation Techniques*, 2004.

[16] D. Liu and Y. Lee. Pfair scheduling of periodic tasks with allocation constraints on multiple processors. *Proc. of the 12th Int'l Workshop on Parallel and Distributed Real-Time Sys.*, 2004.

[17] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. *Proc. of the 20th Real-Time Sys. Symp.*, pp. 294–303, 1999.

[18] S. Parekh, S. Eggers, H. Levy, and J. Lo. Thread-sensitive scheduling for SMT processors. http://www.cs.washington.edu/research/smt/.

[19] J. Renau. SESC website. http://sesc.sourceforge.net.

[20] S. Shankland and M. Kanellos. Intel to elaborate on new multicore processor. http://news.zdnet.co.uk/hardware/chips/0,39020354,39116043,00.htm, 2003.

[21] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic job scheduling with priorities for a simultaneous multithreading processor. *Proc. of ACM SIGMETRICS 2002*, 2002.

[22] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. *Proc. of the 34th ACM Symp. on Theory of Comp.*, pp. 189–198, 2002.

[23] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. *Proc. of the 24th Real-Time Sys. Symp.*, 2003.

[24] S. Viswanathan and T. Imielinski. Metropolitan area video-on-demand service using pyramid broadcasting. *IEEE Multimedia Systems*, pp. 197–208, 1996.

Appendix: Detailed Proofs

In this appendix, detailed proofs are given. We begin by providing further technical background on Pfair scheduling [2, 3, 4, 5, 22].

Ideal fluid schedule. Of central importance in Pfair scheduling is the notion of an ideal fluid schedule, which is defined below and depicted in Fig. 6. Let $\mathit{ideal}(T, t_1, t_2)$ denote the processor share (or allocation) that T receives in an ideal fluid schedule in $[t_1, t_2)$. $\mathit{ideal}(T, t_1, t_2)$ is defined in terms of $\mathit{share}(T, u)$, which is the share (or fraction) of slot u assigned to task T. $\mathit{share}(T, u)$ is defined in terms of a similar *per-subtask* function f:

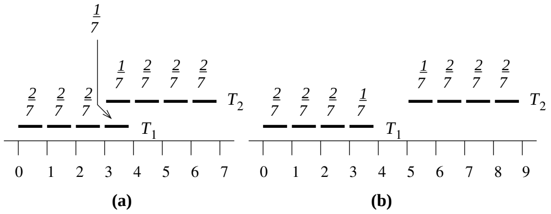


Figure 6: Allocation in an ideal fluid schedule for the first two subtasks of a task T of weight $2/7$. The share of each subtask in each slot of its window $(f(T_i, u))$ is marked. In **(a)**, no subtask is released late; in **(b)**, T_2 is released late. $\mathit{share}(T, 3)$ is either $2/7$ or $1/7$ depending on when subtask T_2 is released.

$$f(T_i, u) = \begin{cases} \left(\left\lfloor \frac{i-1}{wt(T)} \right\rfloor + 1\right) \times wt(T) - (i-1), & u = r(T_i) \\ i - \left(\left\lceil \frac{i}{wt(T)} \right\rceil - 1\right) \times wt(T), & u = d(T_i) - 1 \\ wt(T), & r(T_i) < u < d(T_i) - 1 \\ 0, & \text{otherwise.} \end{cases} \tag{9}$$

Using (9), it follows that $f(T_i, u)$ is at most $wt(T)$. Given f, $\mathit{share}(T, u)$ can be defined as

$$\mathit{share}(T, u) = \textstyle\sum_i f(T_i, u), \tag{10}$$

and then $\mathit{ideal}(T, t_1, t_2)$ as $\sum_{u=t_1}^{t_2-1} \mathit{share}(T, u)$. The following is proved in [22] (see Fig. 6).

$$(\forall u \geq 0 :: \mathit{share}(T, u) \leq wt(T)) \tag{11}$$

Lag in an actual schedule. The difference between the total processor allocation that a task receives in the fluid schedule and in an actual schedule $\mathcal{S}$ is formally captured by the concept of *lag*. Let $actual(T, t_1, t_2, \mathcal{S})$ denote the total actual allocation that T receives in $[t_1, t_2)$ in $\mathcal{S}$. Then, the *lag of task T at time t* is

$$\begin{aligned} lag(T, t, \mathcal{S}) &= ideal(T, 0, t) - actual(T, 0, t, \mathcal{S}) \\ &= \textstyle\sum_{u=0}^{t-1} share(T, u) - \sum_{u=0}^{t-1} \mathcal{S}(T, u). \end{aligned} \tag{12}$$

(For conciseness, when unambiguous, we leave the schedule implicit and use $lag(T, t)$ instead of $lag(T, t, \mathcal{S})$.) A schedule for a GIS task system is said to be *Pfair* iff

$$(\forall t, T \in \tau :: -1 < lag(T, t) < 1). \tag{13}$$

Informally, each task's allocation error must always be less than one quantum. The release times and deadlines in (1) are assigned such that scheduling each subtask in its window is sufficient to ensure (13). Letting $0 \leq t' \leq t$, from (12), we have

$$lag(T, t+1) = lag(T, t) + share(T, t) - \mathcal{S}(T, t), \tag{14}$$

$$lag(T, t+1) = lag(T, t') + ideal(T, t', t+1) - actual(T, t', t+1). \tag{15}$$

Another useful definition, the total lag for a task system τ in a schedule $\mathcal{S}$ at time t, $LAG(\tau, t)$, is given by

$$LAG(\tau, t) = \textstyle\sum_{T \in \tau} lag(T, t). \tag{16}$$

Letting $0 \leq t' \leq t$, from (14)–(16), we have

$$LAG(\tau, t+1) = LAG(\tau, t) + \textstyle\sum_{T \in \tau}(share(T, t) - \mathcal{S}(T, t)), \tag{17}$$

$$LAG(\tau, t+1) = LAG(\tau, t') + ideal(\tau, t', t+1) - actual(\tau, t', t+1). \tag{18}$$

Active tasks. It is possible for a GIS (or IS) task to have no eligible subtasks and a share of zero during certain time slots, if subtasks are absent or released late. Tasks with and without subtasks at time t are distinguished using the following definition of an *active* task.

Definition 1: A GIS task U is *active* at time t if it has a subtask U_j such that $r(U_j) \leq t < d(U_j)$.

Task classification. Tasks in τ may be classified as follows with respect to a schedule $\mathcal{S}$ and time t.[*]

$A(t)$**:** Set of all tasks that are scheduled at t.

$B(t)$**:** Set of all tasks that are not scheduled, but are active at t.

$I(t)$**:** Set of all tasks that are neither active nor are scheduled at t.

[*] For brevity, we let the task system τ and schedule $\mathcal{S}$ be implicit in these definitions.

$A(t)$, $B(t)$, and $I(t)$ form a partition of τ, *i.e.*,

$$(A(t) \cup B(t) \cup I(t) = \tau) \wedge (A(t) \cap B(t) = B(t) \cap I(t) = I(t) \cap A(t) = \emptyset). \tag{19}$$

This classification of tasks is illustrated in Fig. 7. From Def. 1, (9), and (10) we have the following.

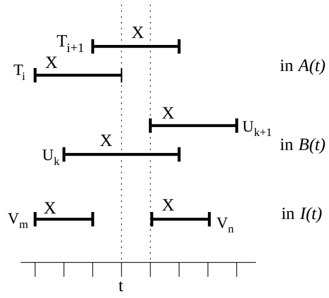


Figure 7: Task classification at time t. Windows of two consecutive subtasks of three GIS tasks T, U, and V are depicted. The slot in which each subtask is scheduled is indicated by an "X." Because subtask T_{i+1} is scheduled at t, $T \in A(t)$. No subtask of U is scheduled at t. However, because the window of U_k overlaps slot t, U is active at t, and hence, $U \in B(t)$. Task V is neither scheduled at t, nor is it active at t. Thus, $V \in I(t)$.

$$(\forall T : T \in I(t) :: share(T, t) = 0) \tag{20}$$

Subtask boundary bit. Each subtask T_i is associated with a bit, denoted $b(T_i)$, defined by (21). From (1), it can be verified that if $\theta(T_i) = \theta(T_{i+1})$, then $b(T_i) = d(T_i) - r(T_{i+1})$. Therefore, $b(T_i)$ determines if the PF-window of T_i can overlap that of T_{i+1}. In Fig. **??**, $b(T_2) = 1$, while $b(T_3) = 0$. Therefore, the PF-window of T_2 overlaps T_3's when $\theta(T_3) = \theta(T_2)$ as in insets (a), (b), and (d).

$$b(T_i) = \left\lceil \frac{i}{wt(T)} \right\rceil - \left\lfloor \frac{i}{wt(T)} \right\rfloor. \tag{21}$$

Tie-break parameters of PD^2. PD^2 uses two tie-break parameters to resolve ties among subtasks with the same deadline. The b-bit given by (21) is the first tie-break parameter. The second tie-break parameter called the "group deadline," is needed in systems with *heavy* tasks. It is easy to show that all the windows of a heavy task with weight in the range $[1/2, 1)$ are of length two or three. For such tasks, the group deadline marks the end of a sequence of windows of length two. Consider a sequence $T_i \cdots T_j$ of subtasks of a heavy periodic task T such that $b(T_k) = 1$, $|w(T_{k+1})| = 2$ for all $i \leq k < j$. Then, scheduling T_i in its last slot forces the other subtasks in this sequence to be scheduled in their last slots, as well. For example, in Fig. 8(a), scheduling T_3 in slot 4 forces T_4 and T_5 to be scheduled in slots 5 and 6, respectively. A group deadline corresponds to a time by which any such "cascade" of scheduling decisions must end. Formally, it is a time t such that either $(t = d(T_i) \wedge b(T_i) = 0)$ or $(t + 1 = d(T_i) \wedge |w(T_i)| = 3)$ for some subtask T_i. The task in Fig. 8(a) has group deadlines at times 4, 8, and 11.

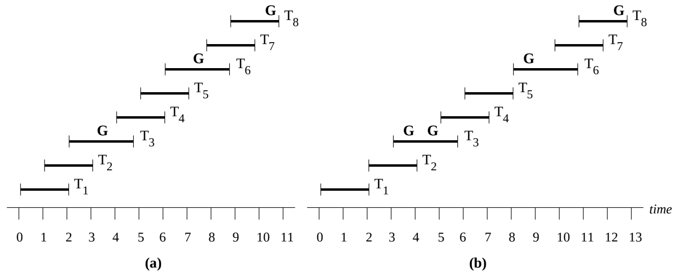


Figure 8: **(a)** Group deadlines of subtasks of a periodic task T with weight $8/11$. Slots that correspond to group deadlines are marked with a "G." The group deadlines of T_1 and T_2 are at time 4, and those of T_3 – T_5 and T_6 – T_8 are at times 8 and 11, respectively. **(b)** Group deadlines of subtasks of an IS task T. In this example, T_2 and T_6 are released late. Nevertheless, the group deadline of T_1 is still 4. However, the group deadline of T_2 is at time 5. Similarly, though T_6 is released one time unit late, the group deadlines of T_3 – T_5 are at time 9 (computed under the assumption that T_6 would be released in time). The group deadlines of T_6 – T_8 are at time 13.

We let $D(T_i)$ denote the group deadline of subtask T_i. If T is heavy, then $D(T_i) = (\min u : u \geq d(T_i) \wedge u$ is a group deadline of $T)$. In Fig. 8(a), $D(T_1) = 4$ and $D(T_6) = 11$. If T is light, then $D(T_i) = d(T_i) + b(T_i)$. If T is an IS task, then T_i's group deadline is computed assuming that all future subtasks are released as early as possible, regardless of how the subtasks are actually released. Fig. 8(b) shows an example with adequate explanation.

PD^2 priority definition. If T_i is ready, then the priority of T_i at time t is given by $(d(T_i), b(T_i), D(T_i))$. Priorities are ordered by the following relation.

$$(d, b, D) \preceq (d', b', D') \equiv (d > d') \vee ((d = d') \wedge (b > b')) \vee \; ((d = d') \wedge (b = b') \wedge (D \geq D')) \tag{22}$$

If T_i and U_j are both ready at time t, then the priority of T_i is at least that of U_j, denoted $T_i \preceq U_j$, if $(d(T_i), b(T_i), D(T_i)) \preceq (d(U_j), b(U_j), D(U_j))$ holds. If $(d(U_j), b(U_j), D(U_j)) \not\preceq (d(T_i), b(T_i), D(T_i))$ holds in addition, then the priority of T_i is strictly greater than that of U_j, denoted $T_i \prec U_j$.

The next definition identifies the last-released subtask at t of any task U.

Definition 2: Subtask U_j is the *critical subtask of* U *at* t iff $e(U_j) \leq t < d(U_j)$ holds, and no other subtask U_k of U, where $k > j$, satisfies $e(U_k) \leq t < d(U_k)$. For example, in Fig. 7, T_{i+1} is the critical subtask of T at both $t - 1$ and t, and U_{k+1} is that of U at $t + 1$.

Lemma 2 *Let T_i be a subtask of a GIS task T such that $b(T_i) = 1$ and let T_k be the successor of T_i. If $d(T_i) \leq u < D(T_i)$ and $u \leq r(T_k)$, then $share(T, d(T_i) - 1) + share(T, u) \leq wt(T)$.*

Displacements. In our proof, we consider task systems obtained by removing subtasks. If $\mathcal{S}$ is a schedule for a GIS task system τ, then removing a subtask from τ results in another GIS system τ', and may cause other subtasks to shift earlier in $\mathcal{S}$, resulting in a schedule $\mathcal{S}'$ that is valid for τ'. Such a shift is called a *displacement* and is denoted by a 4-tuple $\langle X^{(1)}, t_1, X^{(2)}, t_2 \rangle$, where $X^{(1)}$ and $X^{(2)}$ represent subtasks. This is equivalent to saying that subtask $X^{(2)}$ originally scheduled at t_2 in $\mathcal{S}$ displaces subtask $X^{(1)}$ scheduled at t_1 in $\mathcal{S}$. A displacement $\langle X^{(1)}, t_1, X^{(2)}, t_2 \rangle$ is *valid* iff $e(X^{(2)}) \leq t_1$. Because there can be a cascade of shifts, we may have a chain of displacements. This chain is represented by a sequence of 4-tuples. For an example of a displacement chain, refer to Fig. 9.

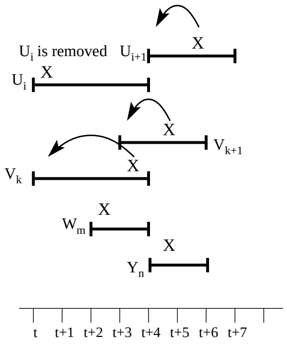


Figure 9: Illustration of displacements. If U_j, scheduled at time t, is removed from the task system, then some subtask that is eligible at t, but scheduled later, can be scheduled at t. In this example, it is subtask V_k (scheduled at $t + 3$). This displacement of V_k results in two more displacements, those of V_{k+1} and U_{i+1}, as shown. Thus, there are three displacements in all: $\Delta_1 = (U_i, t, V_k, t+3)$, $\Delta_2 = (V_k, t+3, V_{k+1}, t + 4)$, and $\Delta_3 = (V_{k+1}, t + 4, U_{i+1}, t + 5)$.

The next lemma concerns displacements and is proved in [22]. It states the priority of every subtask that gets displaced from where it is originally scheduled when some subtask T_i is removed is at most that of T_i.

Lemma 3 (from [22]**)** *Let $X^{(1)}$ be a subtask that is removed from τ, and let the resulting chain of displacements in a PD^2 schedule for τ be $\Delta_1, \Delta_2, \ldots, \Delta_k$, where $\Delta_i = \langle X^{(i)}, t_i, X^{(i+1)}, t_{i+1} \rangle$. Then $(d(X^{(1)}), b(X^{(1)}), D(X^{(1)})) \preceq (d(X^{(i)}), b(X^{(i)}), D(X^{(i)}))$ for all $i \in [1, k]$.*

Proof of Theorem 1

We now prove that W_{sch}, given by (7), is a sufficient scheduling weight for γ to ensure that all component-task deadlines are met. It can be verified that $\left(\frac{W_{\max}-f}{1+f-W_{\max}}\right) \times f$ is at most $1-f$. Therefore, Δ_f is at most $1-f$, and hence $W_{sch} = I + f + \Delta_f$ is at most $I+1$. If W_{sch} is $I+1$, then γ will be allocated exactly $I+1$ processors in every slot, and hence, correctness for component tasks follows from the optimality of PD^2 [22]. Similarly, no component task deadlines will be missed when $f = 0$. Therefore, we only need to consider the case

$$f > 0 \ \wedge \ \Delta_f < 1 - f. \tag{23}$$

Let F denote the fictitious synchronous, periodic task F of weight $f + \Delta_f$ associated with γ. If $\mathcal{S}$ denotes the root-level schedule, then because PD^2 is optimal, by (13), the following holds. (We assume that the total number of processors available at the root is at least the total weight of all the megatasks after inflation and any free tasks.)

$$(\forall t :: -1 < lag(F, t, \mathcal{S}) < 1) \tag{24}$$

Our proof is by contradiction. Therefore, we assume that t_d and γ defined as follows exist.

Definition 3: t_d is the earliest time that the component task system of any megatask misses a deadline under PD^2, *i.e.*, the component task system of some megatask misses a deadline at t_d and there does not exist a megatask whose component task system misses a deadline prior to t_d, when the megatask itself is scheduled by the root-level PD^2 scheduler according to its scheduling weight.

Definition 4: γ is a megatask with the following properties.

(T1) t_d is the earliest time that a component-task deadline is missed in $\mathcal{S}_\gamma$, a PD^2 schedule for the component tasks of γ.

(T2) The component task system of no megatask satisfying (T1) releases fewer subtasks in $[0, t_d)$ than that of γ.

Our setup here is similar to that used by Srinivasan and Anderson in the optimality proof of PD^2 [22]. The only difference is in the number of processors available for scheduling the task systems under consideration. Whereas the number of processors available to the system is the same in every slot and is at least equal to the total system utilization in their case, this number is not uniform across slots and can be either the floor or the ceiling of the total weight of the component tasks in our case.

Because of the similarity in the setup, some of the properties proved in [22] concerning a task system assumed to be missing a deadline under PD^2 also hold for the component task system of γ. We therefore borrow the properties that are relevant to us and provide some intuitive explanation. In what follows, $\mathcal{S}$ denotes the root-level schedule for the task system to which γ belongs. The total system LAG of the component task system of γ with respect to $\mathcal{S}_\gamma$ (which, as mentioned above, is the PD^2 schedule for the component tasks of γ in which a deadline is missed at t_d) is denoted $LAG(\gamma, t, \mathcal{S}_\gamma)$ and is given by the sum of the lags of its component tasks, *i.e.*,

$$LAG(\gamma, t, \mathcal{S}_\gamma) = \textstyle\sum_{T \in \gamma} lag(T, t, \mathcal{S}_\gamma). \tag{25}$$

By (11), the total processor share allocated to tasks in γ in an ideal schedule is given by

$$share(\gamma, t, \mathcal{S}_\gamma) = \sum_{T \in \gamma} share(T, t, \mathcal{S}_\gamma) \leq \sum_{T \in \gamma} wt(T) = I + f. \tag{26}$$

Because γ is scheduled with a weight of W_{sch} (refer to (7)), the corresponding fictitious task F is assigned a weight of $f + \Delta_f$ by the top-level scheduler, and hence, receives an allocation of $f + \Delta_f$ in each slot in an ideal schedule. Before beginning the proof, we introduce some terms.

Tight and non-tight slots. A time slot in which I (resp., $I + 1$) processors are allocated to γ is said to be a *tight* (resp., *non-tight*) slot for γ. Equivalently, if t is a non-tight (tight) slot for γ, then F is allocated (not allocated) in $\mathcal{S}$. In Fig. 3, slots 0 and 2 are non-tight, whereas slot 1 is tight.

Holes. If less than I (resp., $I + 1$) tasks are scheduled in a tight (resp., non-tight) slot t in $\mathcal{S}_\gamma$, then one or more processors are idle at t. If k processors assigned to γ are idle at t, then there are k *holes* in $\mathcal{S}_\gamma$ at t.

Definition 5: A time slot in which every processor allocated to γ (in that slot) is idled is called a *fully-idle slot* for γ. A time slot in which every processor is busy (*i.e.*, a slot without holes), is called a *busy slot*, and one that is neither fully-idle nor busy is called a *partially-idle slot*. An interval $[t_1, t_2)$ in which every slot is fully-idle (resp., partially-idle, busy) is called a *fully-idle* (resp., *partially-idle*, *busy*) *interval*.

Lemma 4 (from [22]) *The following properties hold for γ and $\mathcal{S}_\gamma$.*

(**a**) *For all T_i in γ, $d(T_i) \leq t_d$.*
(**b**) *Exactly one subtask of γ misses its deadline at t_d.*
(**c**) $LAG(\gamma, t_d, \mathcal{S}_\gamma) = 1$.
(**d**) *There are no holes in slot $t_d - 1$.*

Parts (a) and (b) follow from (T2). Part (c) follows from part (b). Part (d) holds because it can be shown that the subtask missing its deadline can otherwise be scheduled at $t_d - 1$. By Lemma 4(c) and (24), we have the following.

$$LAG(\gamma, t_d, \mathcal{S}_\gamma) > lag(F, t_d, \mathcal{S}) \tag{27}$$

Overview of the proof. Because $LAG(\gamma, 0, \mathcal{S}_\gamma) = lag(F, 0, \mathcal{S}) = 0$, (27) implies the following. (Informally, it states that there exists a slot across which the LAG of the tasks in γ becomes larger than the lag of F.)

$$(\exists u : u < t_d :: LAG(\gamma, u) \leq lag(F, u)^{\dagger} \wedge \ LAG(\gamma, u + 1) > lag(F, u + 1)) \tag{28}$$

In the schedule in Fig. 3, $t_d = 12$ and $LAG(\gamma, 12) = 1 > 1/2 = lag(F, 12)$. Also, $LAG(\gamma, t) = lag(F, t)$, for $0 \leq t \leq 8$, and $LAG(\gamma, t) > lag(F, t)$, for $9 \leq t \leq 12$, *i.e.*, the lag inequality between γ and F is violated across slot 8. However, it should be noted that the deadline miss of Fig. 3 is due to the use of the ideal weight W_{sum} in scheduling γ. Because our goal is to show that no deadlines can be missed if γ is scheduled using W_{sch}, we show that for every u as defined in (28), (unlike in Fig. 3) there exists

[†] In the rest of this paper, LAG within γ and the lag of F should be taken to be with respect to $\mathcal{S}_\gamma$ and $\mathcal{S}$, respectively.

a time u', where $u+1 < u' \leq t_d$ such that $LAG(\gamma, u') \leq lag(F, u')$ (*i.e.*, we show that the lag inequality is restored by t_d), and thereby derive a contradiction to Lemma 4(c), and hence, to our assumption that γ misses a deadline at t_d.

The next lemma identifies the presence of holes in slot t as a necessary condition for the lag inequality $LAG(\gamma, t) \leq lag(F, t)$ to be violated across t. An example can be found in Fig. 3. Here, the LAG of the tasks in γ is higher than the lag of F for the first time at time 9, and though slot 8 is non-tight, only one task of γ is scheduled there. Hence, there is a hole in slot 8. Informally, the lemma holds because if there is no hole in slot t, then the difference between the allocations in the ideal and actual schedules for γ would be at most that for F, and hence, the increase in LAG cannot be higher than the increase in lag. This lemma is analogous to one that is heavily used in other work on Pfair scheduling. It should also be noted that, in the case of megatasks, the LAG of the tasks in γ can increase across a tight slot even if there are no holes (*e.g.*, in Fig. 3, $LAG(\gamma, 6) > LAG(\gamma, 5)$, but there is no hole in slot 5), but it is guaranteed not to exceed the lag of F.

Lemma 5 *If $LAG(\gamma, t) \leq lag(F, t)$ and $LAG(\gamma, t+1) > lag(F, t+1)$, then there is at least one hole in slot t.*

Proof: We prove the lemma by proving the contrapositive, *i.e.*, we show that if there are no holes in slot t and

$$LAG(\gamma, t) \leq lag(F, t) \tag{29}$$

holds, then $LAG(\gamma, t+1) \leq lag(F, t+1)$ follows. We consider the following two cases.

Case 1: *t* is a tight slot. Because there are no holes in t, we have $\sum_{T \in \gamma} \mathcal{S}_\gamma(T, t) = I$. Hence, by (18) and (26), $LAG(\gamma, t+1) = LAG(\gamma, t) + share(\gamma, t) - \sum_{T \in \gamma} \mathcal{S}_\gamma(T, t) = LAG(\gamma, t) + f$ holds, which by (29), implies that $LAG(\gamma, t+1) \leq lag(F, t) + f$ holds. Also, because t is a tight slot, F is not scheduled in t. Hence, by (14), we have, $lag(F, t+1) = lag(F, t) + f + \Delta_f$. Thus, $LAG(\gamma, t+1) \leq lag(F, t+1) - \Delta_f \leq lag(F, t+1)$ follows.

Case 2: *t* is a non-tight slot. For this case, we have $\sum_{T \in \gamma} \mathcal{S}_\gamma(T, t) = I + 1$, and hence, by (18) and (26), $LAG(\gamma, t+1) \leq LAG(\gamma, t) + f - 1$ holds. Because t is non-tight, F is scheduled at t, and hence, by (14), $lag(F, t+1) = lag(F, t) + f + \Delta_f - 1$ holds, which by (29) and the previous expression for $LAG(\gamma, t+1)$ implies $LAG(\gamma, t+1) \leq lag(F, t+1)$. ■

We next state three lemmas that we borrow from [22].

Lemma 6 (from [22]) *Let $t < t_d - 1$ be a slot with at least a hole in $\mathcal{S}_\gamma$, let U be any task in $B(t)$, and let U_j be a subtask of U that is scheduled before t. If U_j is the critical subtask of U at t, then, $d(U_j) = t+1$ and $b(U_j) = 1$. Else, $d(U_j) < t$.*

Lemma 7 (from [22]) *Let $t < t_d - 1$ be a slot with at least a hole in $\mathcal{S}_\gamma$, let U be any task in $B(t)$, and U_j its critical subtask at t. Then, there is a slot with no holes in $[t+1, \min(D(U_j), t_d))$.*

Lemma 8 (from [22]) *Let $t < t_d - 1$ be a slot with at least a hole in $\mathcal{S}_\gamma$, let T be any task in $A(t)$, and T_i its subtask scheduled at t. Then, $d(T_i) = t+1$ and $b(T_i) = 1$.*

The next lemma bounds the total ideal allocation in the interval $[t, u+1)$, where there is at least one hole in every slot in $[t, u)$, and u is a busy slot. For an informal proof of this lemma, refer to Fig. 10. As shown in this figure, if task T is in $B(t)$ (as defined in Sec. 2), then no subtask of T with release time prior to t can have its deadline later than $t+1$. Otherwise, because there is a

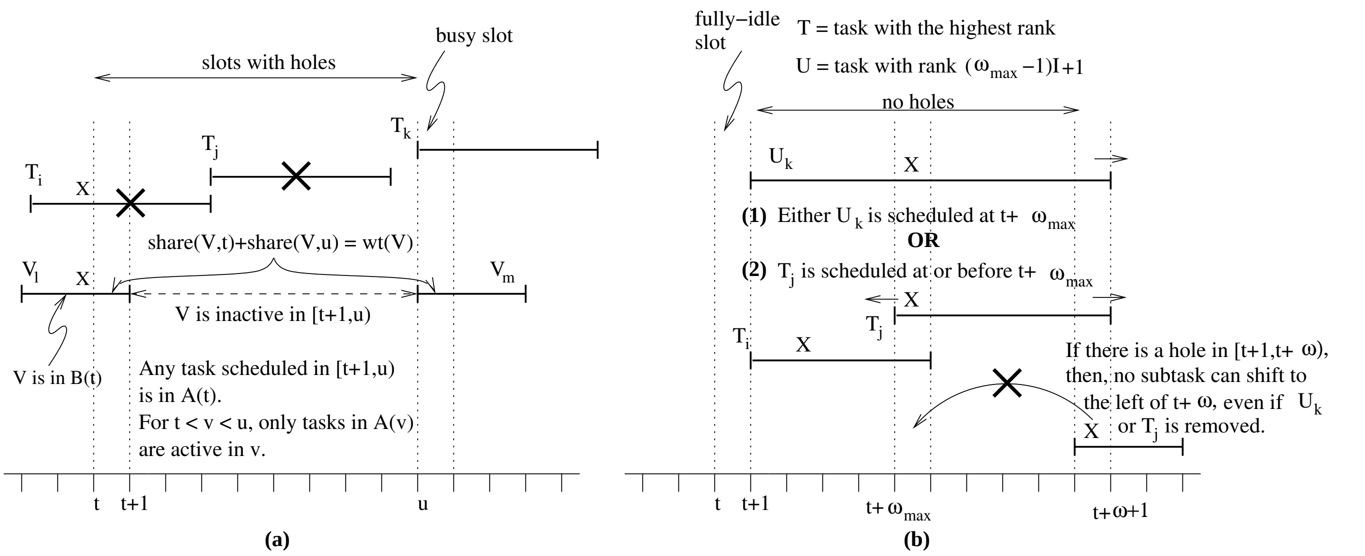

slots with holes
busy slot
T$_k$
T$_j$
T$_i$
X
share(V,t)+share(V,u) = wt(V)
V$_l$
X
V$_m$
V is inactive in [t+1,u)
V is in B(t)
Any task scheduled in [t+1,u) is in A(t).
For t < v < u, only tasks in A(v) are active in v.
t
t+1
u
(a)
fully−idle slot
T = task with the highest rank
U = task with rank (ω$_{max}$ −1)I+1
no holes
U$_k$
X
(1) Either U$_k$ is scheduled at t+ ω$_{max}$
OR
(2) T$_j$ is scheduled at or before t+ ω$_{max}$
X
T$_j$
T$_i$
X
If there is a hole in [t+1,t+ ω), then, no subtask can shift to the left of t+ ω, even if U$_k$ or T$_j$ is removed.
X
t
t+1
t+ ω$_{max}$
t+ω+1
(b)

Figure 10: The slot in which a subtask is scheduled is indicated with an "X." **(a)** Lemma 9. If T is in $B(t)$, subtasks like T_i or T_j cannot exist. Also, a task in $B(t)$ is inactive in $[t+1, u)$. **(b)** Lemma 11. A subtask like U_k or T_j exists. For simplicity, both these subtasks are depicted with equal deadlines. The value of ω may differ for the two cases.

hole in every slot in $[t+1, u)$, removing such a subtask would not result in any subtask scheduled at or after u to shift to the left, and hence, the deadline miss at t_d would not be eliminated, contradicting (T2). Similarly, no subtask of T can have its release time in $[t+1, u)$, and thus, no subtask in $B(t)$ is active in $[t+1, u)$. Furthermore, it can be shown that the total ideal allocation to T in slots t and u is at most $wt(T)$, using which, it can be shown that the total ideal allocation to γ in slots t and u is at most $I + f$ (because this bounds from above the total weight of tasks in $B(t) \cup A(t)$) plus the cumulative weights of tasks scheduled in t (*i.e.*, tasks in A(t)), which is at most $|A(t)|W_{\max}$. Finally, it can be shown that the ideal allocation to γ in a slot s in $[t+1, u)$ is at most $|A(s)|W_{\max}$. Adding all of these values, we get the value indicated in the lemma.

Lemma 9 *Let $t < t_d - 1$ be a fully- or partially-idle slot in $\mathcal{S}_\gamma$ and let $u < t_d$ be the earliest busy slot after t* (i.e.*, $t + 1 \le u < t_d$*) *in $\mathcal{S}_\gamma$. Then, $ideal(\gamma, t, u+1) = \sum_{s=t}^{u} \sum_{T \in \gamma} share(T, s) \le I + f + \sum_{s=t}^{u-1} |A(s)|W_{\max}$.*

Proof: We make use of the above three lemmas to prove this lemma. From the statement of the lemma we have the following.

(H) There is at least a hole in every slot in $[t, u)$.

We first make Claims 1 and 2 below.

Claim 1 *Only tasks in $A(t)$ are active in $[t+1, u)$.*

Proof: To prove this claim, we first show that the following holds.

(C) Only tasks in $A(t)$ are scheduled in $[t+1, u)$.

Assume that (C) does not hold. Hence, there exists a $T \notin A(t)$ such that t', where $t + 1 \le t' < u$ is the earliest slot in $[t+1, u)$ in which T is scheduled. Let T_i be T's subtask scheduled at t'. Because (H) holds, by Lemma 8, $d(T_i) = t'+1$ and $b(T_i) = 1$ hold. Hence, by (21), $wt(T) \ne 1$. Therefore, by (1) and (2), $r(T_i) \le d(T_i) - 2 = t' - 1$, and hence, by (**??**), $e(T_i) \le t' - 1 \le t$. However, by the definition of t' and (H), there is a hole in $t' - 1$. By our assumption, T

is not scheduled in $[t+1, t')$, and because $T \notin A(t)$, it is not scheduled in t either. Thus, T is not scheduled in $t'-1$. Hence, because $e(T_i) \leq t'-1$ holds, T_i should be scheduled in $t'-1$, which is a contradiction. Thus, only tasks that are scheduled at t could be scheduled in $[t+1, u)$.

We next show that only tasks in $A(t)$ are active in $[t+1, u)$. Assume to the contrary and let U be a task that is active in $[t+1, u)$ but is not in $A(t)$, which by (C), implies that U is not scheduled in any slot in $[t, u)$. Hence, if U is active in t', where $t+1 \leq t' < u$, then by Def. 1, there exists a subtask U_j such that $e(U_j) \leq t' < d(U_j)$. Also, because U is not scheduled anywhere in $[t, u)$ and (H) holds, U_j should have been scheduled before t, and hence, $e(U_j) \leq t-1$ holds. That is, we have the following.

$$e(U_j) \leq t-1 \qquad\qquad d(U_j) \geq t'+1 > t+1 \tag{30}$$

By Def. 1, the definition of $I(t)$, and (30), U cannot be in $I(t)$. By our assumption, U is not in $A(t)$, and hence, should be in $B(t)$. But then, because U_j is scheduled before t and there is a hole in t, by Lemma 6, $d(U_j) \leq t+1$, which contradicts (30). Therefore, our assumption that U is not in $A(t)$ is incorrect. □

Claim 2 $(\forall t' : t+1 \leq t' < u :: B(t') = \emptyset)$.

Proof: Assume that the claim does not hold and let $t+1 \leq t' < u$ be a slot such that $B(t') \neq \emptyset$. Let U be any task in $B(t')$. Thus, U is active at $t'$and let U_j be the critical subtask of U at t'. Because U is not scheduled in t', U_j should have been scheduled before t', say at $\hat{t} < t'$. Hence, by Lemma 6,

$$d(U_j) = t'+1 > t+1 \tag{31}$$

holds. Because $d(U_j) = t'+1 > \hat{t}+1$ holds, by the contrapositive of Lemma 8, there is no hole in slot $\hat{t}$, which by (H) implies that $\hat{t} < t$. However, then by (H), Lemma 6 implies that $d(U_j) \leq t+1$, which contradicts (31). The claim follows. □

We next show the following.

(D) $(\forall U \in B(t) :: \mathit{share}(U, t) + \mathit{share}(U, u) \leq \mathit{wt}(U))$.

Let U be any task in $B(t)$ and let U_j be its critical subtask at t. Then, by Lemma 6, $d(U_j) = t+1$ and $b(U_j) = 1$. By Lemma 7, there is a slot without a hole in $[t+1, \min(t_d, D(U_j)))$. Hence, $u \leq \min(t_d - 1, D(U_j) - 1)$ holds. Also, by Claim 1, U is inactive over $[t+1, u)$. Therefore, by Def. 1, $r(U_k) \geq u$ holds for U_j's successor U_k. Also, $u \geq t+1 = d(U_j)$ holds. Hence, by Lemma 2, $\mathit{share}(U, d(U_j) - 1) + \mathit{share}(U, u)$, *i.e.*, $\mathit{share}(U, t) + \mathit{share}(U, u)$ is at most $\mathit{wt}(U)$.

We are now ready to prove the lemma.

$$\sum_{s=t}^{u} \sum_{T \in \gamma} \mathit{share}(T, s) = \sum_{s=t}^{u} \left(\sum_{T \in A(t)} \mathit{share}(T, s) + \sum_{T \in B(t)} \mathit{share}(T, s) + \sum_{T \in I(t)} \mathit{share}(T, s) \right) \qquad \{\text{by (19)}\}$$

$$= \sum_{s=t}^{u} \sum_{T \in A(t)} \mathit{share}(T, s) + \sum_{T \in B(t)} \mathit{share}(T, t) + \sum_{T \in B(t) \cup T \in I(t)} \mathit{share}(T, u)$$

$\{$Because only tasks in $A(t)$ are active in $[t+1, u)$ by Claim 1.

By (20), $\mathit{share}(T, s) = 0$ for all $T \in I(s)$, for all s.$\}$

$$= \sum_{s=t}^{u-1} \sum_{T \in A(t)} \mathit{share}(T, s) + \sum_{T \in B(t)} \mathit{share}(T, t) + \sum_{T \in A(t) \cup B(t) \cup I(t)} \mathit{share}(T, u)$$

$$= \sum_{T \in A(t)} \mathit{share}(T, t) + \sum_{s=t+1}^{u-1} \left(\sum_{T \in A(s)} \mathit{share}(T, s) + \sum_{T \in A(t) \setminus A(s)} \mathit{share}(T, s) \right) + \sum_{T \in B(t)} \mathit{share}(T, t) + \sum_{T \in \gamma} \mathit{share}(T, u)$$

$\{$By Claim 1, $A(s) \subseteq A(t)$, for $t + 1 \leq s < u$.$\}$

$$= \sum_{s=t}^{u-1} \sum_{T \in A(s)} \mathit{share}(T, s) + \sum_{T \in B(t)} \mathit{share}(T, t) + \sum_{T \in \gamma} \mathit{share}(T, u)$$

$\{$By Claim 2, $B(s) = \emptyset$, for $t + 1 \leq s < u$, and hence, every $T \in A(t) \setminus A(s)$ is in $I(s)$.$\}$

$$= \sum_{s=t}^{u-1} \sum_{T \in A(s)} \mathit{share}(T, s) + \sum_{T \in B(t)} (\mathit{share}(T, t) + \mathit{share}(T, u)) + \sum_{T \in \gamma \setminus B(t)} \mathit{share}(T, u)$$

$$\leq \sum_{s=t}^{u-1} \sum_{T \in A(s)} wt(T) + \sum_{T \in B(t)} (\mathit{share}(T, t) + \mathit{share}(T, u)) + \sum_{T \in \gamma \setminus B(t)} wt(T) \qquad \{\text{by (11)}\}$$

$$\leq \sum_{s=t}^{u-1} \sum_{T \in A(s)} wt(T) + \sum_{T \in B(t)} wt(T) + \sum_{T \in \gamma \setminus B(t)} wt(T) \qquad \{\text{by Lemma 2}\}$$

$$\leq \sum_{s=t}^{u-1} \sum_{T \in A(s)} W_{\max} + W_{sum}(\gamma) \qquad \{\text{by (4) and (3)}\}$$

$$= \sum_{s=t}^{u-1} |A(s)| W_{\max} + I + f$$

■

The following two lemmas concern fully-idle slots.

Lemma 10 *Let $t < t_d$ be a fully-idle slot in $\mathcal{S}_\gamma$. Then all slots in $[0, t+1)$ are fully idle in $\mathcal{S}_\gamma$.*

Proof: Suppose, to the contrary, that some subtask T_i is scheduled before t. Then, removing T_i from $\mathcal{S}_\gamma$ will not cause any subtask scheduled after t to shift to the left to t or earlier. (If such a displacement to the left occurs, then the displacing subtask should have been scheduled at t even when T_i is included.) Hence, even if every subtask scheduled before t is removed, the deadline miss at t_d cannot be eliminated. This contradicts (T2). ■

The second lemma concerning fully-idle slots says that there is a minimum number of busy slots following the last fully-idle slot in $\mathcal{S}_\gamma$. Informally, the lemma holds because it can be shown that one of the two cases in Fig. 10(b) holds, and hence, if there is a hole in $[t+1, t+\omega)$, then a subtask (like U_k or T_j) with a deadline at or after $t + 1 + \omega$ but scheduled before $t + \omega$ can be removed without causing any subtask scheduled later (at or after $t + \omega$) to shift earlier. Hence, the deadline miss at t_d would not be eliminated, contradicting (T2).

Lemma 11 *Let $t < t_d - 1$ be the last fully-idle slot in $\mathcal{S}_\gamma$ and let ω be as defined in* (6)*. Then, every slot in* $[t+1, t+\omega)$ *is busy. Also,* $t_d \geq t + 1 + \omega$.

Proof: By Lemma 10, every slot in $[0, t+1)$ is fully idle. Hence, $r(T_i) \geq e(T_i) \geq t+1$ holds for every subtask T_i in $\mathcal{S}_\gamma$. By (1), $d(T_i) - r(T_i) \geq \frac{1}{wt(T)}$ holds, and because $r(T_i)$ and $d(T_i)$ are integral, $d(T_i) - r(T_i) \geq \left\lceil \frac{1}{wt(T)} \right\rceil \geq \left\lceil \frac{1}{W_{\max}} \right\rceil$ holds. So, by (5), we have $d(T_i) \geq r(T_i) + \left\lceil \frac{1}{wt(T)} \right\rceil \geq r(T_i) + \omega_{\max}$ for every subtask T_i in $\mathcal{S}_\gamma$. If $W_{\max} = \frac{1}{k}$, where k is an integer greater than zero, then $\left\lceil \frac{j}{W_{\max}} \right\rceil = \left\lfloor \frac{j}{W_{\max}} \right\rfloor = \frac{j}{W_{\max}}$ holds for all $j \in \mathbb{N}$, and hence, by (21), $b(U_j) = 0$ holds for all subtasks U_j of a task U with weight $W_{\max}$. Therefore, if $W_{\max} = \frac{1}{k}$ and $wt(T) < W_{\max}$, then $\left\lceil \frac{1}{wt(T)} \right\rceil > \left\lceil \frac{1}{W_{\max}} \right\rceil$ holds, and hence, either $d(T_i) > r(T_i) + \omega_{\max}$ or $(d(T_i) = r(T_i) + \omega_{\max} \ \wedge \ b(T_i) = 0)$ holds for every subtask T_i in $\mathcal{S}_\gamma$. Thus, we have the following.

$$(\forall T_i \in \mathcal{S}_\gamma :: r(T_i) \geq t+1) \tag{32}$$

$$(\forall T_i \in \mathcal{S}_\gamma :: W_{\max} \neq \frac{1}{k} \ \Rightarrow d(T_i) \geq r(T_i) + \left\lceil \frac{1}{wt(T)} \right\rceil \geq r(T_i) + \omega_{\max}) \tag{33}$$

$$(\forall T_i \in \mathcal{S}_\gamma :: W_{\max} = \frac{1}{k} \ \Rightarrow (d(T_i) = r(T_i) + \omega_{\max} \ \wedge \ b(T_i) = 0) \ \vee \ d(T_i) \geq r(T_i) + \left\lceil \frac{1}{wt(T)} \right\rceil > r(T_i) + \omega_{\max}) \tag{34}$$

(32) – (34) above imply (35) and (36) below.

$$(\forall T_i \in \mathcal{S}_\gamma :: W_{\max} \neq \frac{1}{k} \ \Rightarrow t + \omega_{\max} \leq d(T_i) - 1) \tag{35}$$

$$(\forall T_i \in \mathcal{S}_\gamma :: W_{\max} = \frac{1}{k} \ \Rightarrow ((t + \omega_{\max} = d(T_i) - 1 \wedge b(T_i) = 0) \ \vee \ t + \omega_{\max} < d(T_i) - 1)) \tag{36}$$

Let t' be a slot in $[t+1, t+\omega_{\max})$. By the statement of this lemma and Lemma 10, t' is not fully idle, and let U_j be a subtask scheduled in t'. Then, by (35)–(36), $d(U_j) \geq t + 1 + \omega_{\max}$ holds, and hence, by Lemma 8, there cannot a hole in t'. Further, if $W_{\max} = \frac{1}{k}$ and $d(U_j) = t + 1 + \omega_{\max}$, then by (36), $b(U_j) = 0$, and hence, by Lemma 8 again, there cannot be a hole in $t + \omega_{\max}$, either. Define t_e as follows.

$$t_e = \begin{cases} t + 1 + \omega_{\max}, & W_{\max} = \frac{1}{k} \\ t + \omega_{\max}, & W_{\max} \neq \frac{1}{k} \end{cases} \tag{37}$$

Then, by the above discussion, we have the following.

(L) There are no holes in $[t+1, t_e)$.

To establish the lemma, we need to prove that there are no holes in $[t_e, t+\omega)$ and show that $t_d \geq t + \omega + 1$. For this, we make Claims 3–6 below.

Claim 3 *Let U_k be a subtask in $\mathcal{S}_\gamma$ and let U_k's predecessor be also present in $\mathcal{S}_\gamma$. Then,* $d(U_k) \geq t + 2\omega_{\max}$.

Proof: Let U_j be U_k's predecessor in $\mathcal{S}_\gamma$. By (32)–(34), $d(U_j) \geq t + 1 + \omega_{\max}$ holds. Hence, by (2) and (1), $r(U_k) \geq t + \omega_{\max}$ and by (33) and (34), $d(U_k) \geq t + 2 \cdot \omega_{\max}$ holds. □

Claim 4 *There are at least $I \cdot \omega_{\max} + 1$ component tasks in γ if $W_{\max} = \frac{1}{k}$, and at least $I \cdot (\omega_{\max} - 1) + 1$ otherwise.*

Proof: By (3), $\sum_{T \in \gamma} wt(T) = I + f$, and hence,by (4), $\sum_{T \in \gamma} W_{\max} \geq I + f$. Let n denote the number of component tasks in γ. Then, $n \times W_{\max} \geq I + f$ holds. Thus, $n \geq \frac{I+f}{W_{\max}}$. If $W_{\max} = \frac{1}{k}$, then $\left\lceil \frac{1}{W_{\max}} \right\rceil = \frac{1}{W_{\max}}$ is an integer, and because n is an integer and $f > 0$, $n \geq \frac{I}{W_{\max}} + 1 = \omega_{\max} \cdot I + 1$ follows from (5). If $W_{\max} \neq \frac{1}{k}$, then $n \geq \frac{I+f}{W_{\max}} \geq (\left\lceil \frac{1}{W_{\max}} \right\rceil - 1)I + \frac{f}{W_{\max}}$, which by the integral nature of n implies $n \geq (\left\lceil \frac{1}{W_{\max}} \right\rceil - 1)I + 1 = (\omega_{\max} - 1) \cdot I + 1$. □

Claim 5 *If there exists a task that is scheduled more than once in $[t+1, t_e)$, then there are no holes in $[t_e, \bar{\omega})$ and $t_d \geq t + 1 + \bar{\omega}$, where* $\bar{\omega} = \begin{cases} 2\omega_{\max} - 1, & W_{\max} \neq \frac{1}{k} \\ 2\omega_{\max}, & W_{\max} = \frac{1}{k} \end{cases}$.

Proof: Let V be a task that is scheduled more than once in $[t+1, t_e)$. Let V_k, scheduled at t', be the latest subtask of V scheduled before t_e. We consider the following two cases.

Case 1: $W_{\max} \neq \frac{1}{k}$. Because V is scheduled more than once in $[t+1, t_e)$, by Claim 3, $d(V_k) \geq t + 2 \cdot \omega_{\max}$ holds. We next show the following.

(J) There exists a subtask X_m scheduled after t' such that $e(X_m) \leq t'$, $d(X_m) \geq t + 2 \cdot \omega_{\max}$, and the predecessor of X_m, if one exists, is scheduled before t'.

If the removal of V_k from $\mathcal{S}_\gamma$ does not result in any other subtask of γ shifting left to t', then V_k can be removed without eliminating the deadline miss at t_d. This would contradict (T2), and hence, there exists a subtask X_m, scheduled after t', that can shift left to t'. By Lemma 3, $d(X_m) \geq d(V_k) \geq t + 2 \cdot \omega_{\max}$ holds for X_m. Furthermore, because X_m can shift into t', $e(X_m) \leq t'$ holds, and its predecessor, if one exists, should have completed executing by t'. Thus, (J) holds.

Let subtask X_m, as defined above, be scheduled at $\hat{t} > t'$. Then, by (J), there is no hole in $[t', \hat{t})$. Thus, if $\hat{t} \geq t + 2 \cdot \omega_{\max}$, then it implies that there is no hole in $[t', t + 2 \cdot \omega_{\max} - 1)$. If $\hat{t} < t + 2 \cdot \omega_{\max}$, then by Lemma 6, there is no hole in $[\hat{t}, t + 2 \cdot \omega_{\max} - 1)$. Thus, there are no holes in any slot in $[t', t + 2 \cdot \omega_{\max} - 1)$, *i.e.*, in $[t', t + \bar{\omega})$. That $t_d \geq t + 2 \cdot \omega_{\max}$ follows from Lemma 4(a) and the fact that $d(V_k) \geq t + 2 \cdot \omega_{\max}$.

Case 2: $W_{\max} = \frac{1}{k}$. Similar to the proof for Case 1. □

Claim 6 *If no task is scheduled more than once in $[t+1, t_e)$, then there are no holes in $[t_e, t + \hat{\omega})$ and $t_d \geq t + 1 + \hat{\omega}$, where* $\hat{\omega} = \begin{cases} \textit{smallest window length of task of rank } (\omega_{\max} - 1) \cdot I + 1, & W_{\max} \neq \frac{1}{k} \\ \textit{smallest window length of task of rank } \omega_{\max} \cdot I + 1, & W_{\max} = \frac{1}{k} \end{cases}$.

Proof: Let no task be scheduled more than once in $[t+1, t_e)$. Then by (L), at least $(t_e - t - 1) \cdot I$ tasks are scheduled before t_e. We prove the claim for $W_{\max} \neq \frac{1}{k}$. The proof for the other case is similar. Let U_j be a subtask that is scheduled at t_e. Because task U is scheduled for the first time in t_e, subtasks of at least $(t_e - t - 1) \cdot I + 1 = (\omega_{\max} - 1) \cdot I + 1$ tasks are scheduled in $[t+1, t_e+1)$. By Claim 4, there are at least these many tasks in γ. Hence, there exists a subtask X_m scheduled in $[t+1, t_e+1)$, such that the rank of X is at least $(\omega_{\max}-1) \cdot I + 1$. (This is because every subtask in $\mathcal{S}_\gamma$ is released at or after $t+1$, and the relative priorities among subtasks released at $t+1$ will depend on the task weights.) Therefore, by (32), (33), and Lemma 1, the deadline of X_m is at least $d = t + 1 + \hat{\omega}$, where $\hat{\omega} \geq \omega_{\max}$ (because $wt(X) \leq W_{\max}$), and so, by Lemma 8, there can be no hole in t_e, and by Lemma 6, no hole in $[t_e + 1, d - 1)$. Thus, there is no hole in$[t + \omega_{\max}, t +$ (smallest window length of component task with rank $(\omega_{\max} - 1) \cdot I + 1))$. Also, by Lemma 4(a), the presence of X_m implies that $t_d \geq t + 1 + \hat{\omega}$. □

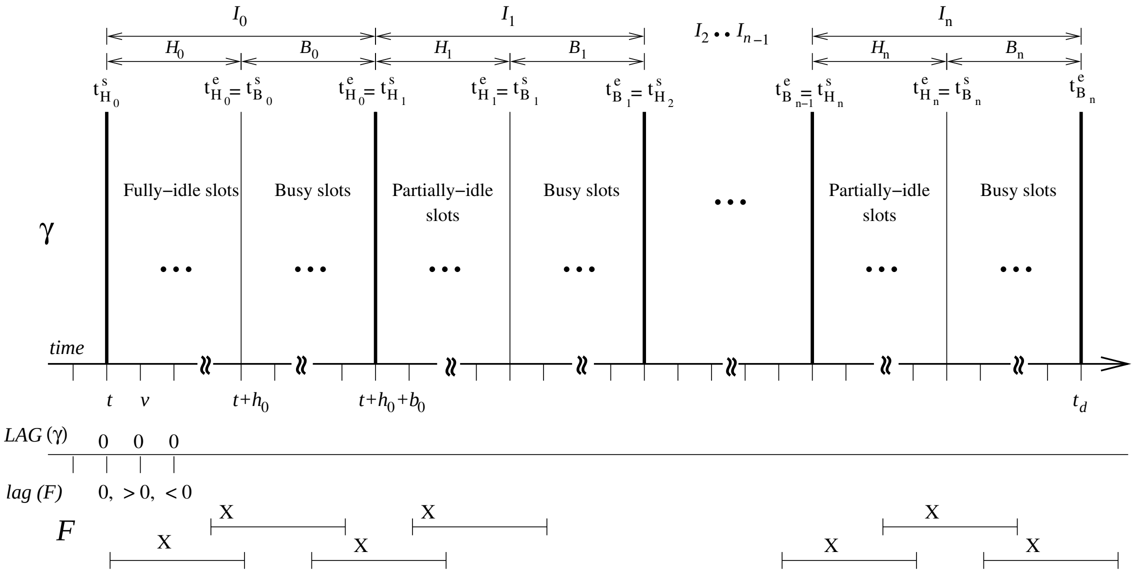


Figure 11: Subintervals of the interval $\mathcal{I} = [t, t_d)$ as explained in Lemma 12. Sample windows and allocations for the fictitious task corresponding to γ (after reweighting) are shown below the time line.

By Claims 5 and 6, $t_d \geq t + 1 + \omega$ and there is no hole in $[t_e, t + \omega)$. Hence by (L), there is no hole in $[t + 1, t + \omega)$. ■

We are now ready to prove the main lemma, which shows that the lag inequality, if violated, is restored by t_d.

Lemma 12 *Let $v < t_d$ be a slot such that $LAG(\gamma, v) \leq lag(F, v)$, but $LAG(\gamma, v + 1) > lag(F, v + 1)$. Then, there exists a time u, where $v + 1 < u \leq t_d$, such that $LAG(\gamma, u) \leq lag(F, u)$.*

Proof: Let $\Delta LAG(\gamma, t_1, t_2)$, where $t_1 < t_2$, denote $LAG(\gamma, t_2) - LAG(\gamma, t_1)$, *i.e.*, the difference between the LAG of the tasks in γ at t_2 and t_1, and let $\Delta lag(F, t_1, t_2)$ be analogously defined. To prove this lemma, it is sufficient to show that $\Delta LAG(\gamma, v, u) \leq \Delta lag(F, v, u)$, where u is as defined in the statement of the lemma. However, when v is fully-idle it is simpler to show that $\Delta LAG(\gamma, t, u) \leq \Delta lag(F, t, u)$, where $t \leq v$ and $lag(F, t) = LAG(\gamma, t)$ hold, and hence, we define t as follows. (As shown in Fig. 11, $LAG(\gamma, t + 1)$ may not exceed $lag(F, t + 1)$, and it is immaterial.)

$$t = \begin{cases} v, & v \text{ is not fully-idle} \\ \max(t' : 0 \leq t' \leq v :: LAG(\gamma, t') = lag(F, t')), & v \text{ is fully-idle} \end{cases} \tag{38}$$

Note that because $LAG(\gamma, 0) = lag(F, 0) = 0$, t is well defined. If v is fully-idle, then by Lemma 10, every slot in $[0, v + 1)$ is fully-idle, and hence, by (38), t either **(i)** is fully-idle, or **(ii)** equals v, and so, by the statement of this lemma and Lemma 5, has at least one hole. Thus, in either case there is at least one hole in t. Hence, by Lemma 4(d), $t < t_d - 1$ holds. Let $\mathcal{I}$ denote the interval $[t, t_d)$. We first partition $\mathcal{I}$ into disjoint subintervals as shown in Fig. 11, where each subinterval is of one of the following three types as defined in Def. 5: **(i)** fully-idle, **(ii)** partially-idle, and **(iii)** busy. Each subinterval is maximal in that it cannot be extended to the right or left without either extending past the interval $\mathcal{I}$ or violating the property that all slots in the subinterval are of the same type.

Because there is at least one hole in t (as discussed above), the first subinterval is either fully-idle or partially-idle. Similarly, because there is no hole in $t_d - 1$, the last subinterval is busy. By Lemma 10, a non fully-idle slot cannot be followed by a fully-idle slot. From (6), it can be verified that $\omega \geq 2$ holds, and hence, by Lemma 11, there is at least one busy slot following the last fully-idle slot in $\mathcal{I}$. Thus, the intermediate subintervals of $\mathcal{I}$ alternate between busy and partially-idle types, in that order. This is illustrated in Fig. 11.

A fully-idle first subinterval, if present, is denoted H_0 and the busy subinterval following it is denoted B_0. The alternating partially-idle and busy subintervals following B_0 are denoted H_k and B_k, respectively, where $1 \leq k \leq n$, and n is the number of such alternating pairs of subintervals. The combined interval $H_k, \ldots, B_k$ is denoted I_k. If v is not fully idle, then H_0 and B_0, and hence, I_0 will be empty. This is expressed formally below.

$$v \text{ is not fully-idle} \Leftrightarrow\ I_0 \text{ is empty} \tag{39}$$

Before proceeding further, we introduce some more notation. $t^s_{H_k}$ (resp., $t^s_{B_k}$) and $t^e_{H_k}$ (resp., $t^e_{B_k}$) denote the starting and ending times, respectively, of subinterval H_k (resp., B_k). h_k and b_k denote the lengths of the subintervals H_k and B_k, respectively. L denotes the cumulative length of I_1 through I_n and L_0 denotes the length of I_0. h^T_k (resp., h^N_k) denotes the number of tight (resp., non-tight) slots in H_k. b^T_k and b^N_k denote corresponding values for B_k. (T and N stand for "tight" and "non-tight," respectively, and are not to be confused with task identifiers.) The cumulative number of tight and non-tight slots in $I_1, \ldots, I_n$ is denoted L^T and L^N, respectively. Finally, P_k denotes the cumulative lengths of subintervals I_0 through I_k. This notation is summarized in Fig. 12.

$$H_k \stackrel{\text{def}}{=} [t^s_{H_k}, t^e_{H_k}) \tag{40}$$

$$B_k \stackrel{\text{def}}{=} [t^s_{B_k}, t^e_{B_k}) \tag{41}$$

$$t \stackrel{\text{def}}{=} \begin{cases} t^s_{H_0}, & v \text{ is not fully-idle} \\ t^s_{H_1}, & v \text{ is fully-idle} \end{cases}$$

$$t_d \stackrel{\text{def}}{=} t^e_{B_n}$$

$$t^e_{H_k} \stackrel{\text{def}}{=} t^s_{B_k}, 0 \leq k \leq n$$

$$t^e_{B_k} \stackrel{\text{def}}{=} t^s_{H_{k+1}}, 0 \leq k \leq n-1$$

$$h_k \stackrel{\text{def}}{=} t^e_{H_k} - t^s_{H_k}, 0 \leq k \leq n$$

$$b_k \stackrel{\text{def}}{=} t^e_{B_k} - t^s_{B_k}, 0 \leq k \leq n$$

$$h^T_k\ (b^T_k) \stackrel{\text{def}}{=} \text{no. of tight slots in } H_k\ (B_k)$$

$$h^N_k\ (b^N_k) \stackrel{\text{def}}{=} \text{no. of non-tight slots in } H_k\ (B_k)$$

$$L \stackrel{\text{def}}{=} \textstyle\sum_{k=1}^{N}(h_k + b_k) \tag{42}$$

$$L_0 \stackrel{\text{def}}{=} h_0 + b_0 \tag{43}$$

$$L^T \stackrel{\text{def}}{=} \textstyle\sum_{k=1}^{N}(h^T_k + b^T_k) \tag{44}$$

$$L^N \stackrel{\text{def}}{=} \textstyle\sum_{k=1}^{N}(h^N_k + b^N_k) \tag{45}$$

$$L^T_0 \stackrel{\text{def}}{=} h^T_0 + b^T_0 \tag{46}$$

$$L^N_0 \stackrel{\text{def}}{=} h^N_0 + b^N_0 \tag{47}$$

$$P_k \stackrel{\text{def}}{=} \textstyle\sum_{i=0}^{k}(h_i + b_i) \tag{48}$$

$$P_{-1} \stackrel{\text{def}}{=} 0 \tag{49}$$

Figure 12: Notation for Lemma 12.

Recall that our goal is to show that there exists a u, where $t+1 < u \leq t_d$, such that $\Delta LAG(\gamma, t, u) \leq \Delta lag(F, t, u)$. Towards that end, we compute the ideal and actual allocations to γ and F in $\mathcal{I}$. By Lemma 9, the total allocation to γ in H_k and the first slot of B_k in the ideal schedule is given by $ideal(\gamma, t^s_{H_k}, t^s_{B_k}+1) \leq \sum_{i=1}^{h_k} |A(t+P_{k-1}+i-1)| \cdot W_{\max} + I + f$. By (26), γ is allocated at most $W_{sum} = I + f$ in each slot in the ideal schedule. Hence, the total ideal allocation in subinterval I_k, which is comprised of H_k and B_k, is given by $ideal(\gamma, t^s_{H_k}, t^e_{B_k}) \leq \sum_{i=1}^{h_k} |A(t+P_{k-1}+i-1)| \cdot W_{\max} + (I+f) + (b_k - 1) \cdot (I+f) = \sum_{i=1}^{h_k} |A(t+P_{k-1}+i-1)| \cdot W_{\max} + b_k \cdot (I+f)$. Thus, the total ideal allocation in $\mathcal{I}$ is given by

$$ideal(\gamma, t, t_d)\ \leq\ \textstyle\sum_{k=0}^{n}\Big(\Big(\sum_{i=1}^{h_k}(|A(t+P_{k-1}+i-1)| \cdot W_{\max})\Big) + b_k \cdot (I+f)\Big). \tag{50}$$

We now determine the number of processors executing tasks of γ in $\mathcal{S}_\gamma$ in $\mathcal{I}$, *i.e.*, the actual allocation to γ in $\mathcal{S}_\gamma$ in $\mathcal{I}$. This number is equal to $|A(t')|$ for a slot t' with a hole, and is equal to I (resp., $I+1$) for a busy tight (resp., non-tight) slot. Hence, the actual allocation to γ in $\mathcal{S}_\gamma$ can be expressed as follows.

$$actual(\gamma, t, t_d) = \textstyle\sum_{k=0}^{n} \left(\left(\sum_{i=1}^{h_k} |A(t + P_{k-1} + i - 1)| \right) \; + \; I \cdot b_k^T \; + \; (I+1) \cdot (b_k - b_k^T) \right) \tag{51}$$

By (50) and (51), we have

$$\begin{aligned}
\Delta LAG(\gamma, t, t_d) \quad &= \quad LAG(\gamma, t_d) - LAG(\gamma, t) \\
&= \quad ideal(\gamma, t, t_d) - actual(\gamma, t, t_d) \qquad\qquad\qquad\qquad \{\text{by (18)}\} \\
&\leq \quad \textstyle\sum_{k=0}^{n} \left(\left(\sum_{i=1}^{h_k} (|A(t + P_{k-1} + i - 1)| \cdot (W_{\max} - 1)) \right) + b_k^T \cdot f + (b_k - b_k^T)(f-1) \right) \qquad (52) \\
&\leq \quad (b_0 - b_0^T)(f-1) + b_0^T \cdot f + \textstyle\sum_{k=1}^{n} \left(\left(\sum_{i=1}^{h_k} (W_{\max} - 1) \right) + b_k^T \cdot f + (b_k - b_k^T)(f-1) \right) \\
&\qquad \{W_{\max} \leq 1, \text{ and hence, (52) is decreases with increasing } |A(t + P_{k-1} + i - 1)|. \text{ However, by (I),} \\
&\qquad H_1, \ldots, H_n \text{ are partially-idle, and hence, } |A(t + P_{k-1} + i - 1)| \geq 1, \text{ for } 1 \leq k \leq n.\} \\
&= \quad b_0^N (f-1) + b_0^T \cdot f + \textstyle\sum_{k=1}^{n} (h_k (W_{\max} - 1) + b_k^T \cdot f + (b_k - b_k^T)(f-1)) \\
&= \quad b_0 \cdot f - b_0^N + \textstyle\sum_{k=1}^{n} (h_k (W_{\max} - 1) + b_k \cdot f - b_k + b_k^T) \\
&= \quad b_0 \cdot f - b_0^N + \textstyle\sum_{k=1}^{n} (h_k \cdot ((W_{\max} - f - 1) + f) + b_k \cdot f - b_k + b_k^T) \\
&= \quad b_0 \cdot f - b_0^N + \textstyle\sum_{k=1}^{n} ((h_k + b_k) \cdot f + h_k \cdot (W_{\max} - f - 1) - b_k^N) \quad \{b_k = b_k^T + b_k^N\} \\
&= \quad b_0 \cdot f - b_0^N + L \cdot f + \textstyle\sum_{k=1}^{n} (h_k \cdot (W_{\max} - f - 1) - b_k^N) \\
&= \quad b_0 \cdot f - b_0^N + L \cdot f + \textstyle\sum_{k=1}^{n} (h_k^T \cdot (W_{\max} - f - 1) + h_k^N \cdot (W_{\max} - f - 1) - b_k^N) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \{h_k = h_k^T + h_k^N\} \\
&\leq \quad b_0 \cdot f - b_0^N + L \cdot f + \textstyle\sum_{k=1}^{n} (h_k^N \cdot (W_{\max} - f - 1) - b_k^N) \qquad \{\text{because } W_{\max} \leq 1\} \\
&= \quad b_0 \cdot f - b_0^N + L \cdot f - L^N + \textstyle\sum_{k=1}^{n} h_k^N \cdot (W_{\max} - f) \qquad\qquad \{\text{by (45)}\} \\
&\leq \quad \begin{cases} (b_0 + L) \cdot f - b_0^N + L^N (W_{\max} - f - 1), & W_{\max} > f \\ (b_0 + L) \cdot f - b_0^N - L^N, & W_{\max} \leq f. \end{cases} \qquad (53)
\end{aligned}$$

Having determined the change in LAG for the tasks in γ across $\mathcal{I}$, we now determine the change in lag for the fictitious task F across the same interval. F receives an allocation of $f + \Delta_f$ in every slot in an ideal system. Hence, by (42) and (43),

$$ideal(F, t, t_d) = \textstyle\sum_{k=0}^{n} (h_k + b_k)(f + \Delta_f) = (L + L_0)(f + \Delta_f) \tag{54}$$

In schedule $\mathcal{S}$, F is allocated in every non-tight slot in $\mathcal{I}$. Hence, by (45) and (47), $actual(F, t, t_d)$ is given by

$$actual(F, t, t_d) = \textstyle\sum_{k=0}^{n} (h_k^N + b_k^N) = L^N + L_0^N. \tag{55}$$

So, by (15), the change in lag of F across $\mathcal{I}$ is given by

$$\Delta lag(F, t, t_d) = lag(F, t_d) - lag(F, t) = ideal(F, t, t_d) - actual(F, t, t_d) = (L + L_0)(f + \Delta_f) - L^N - L_0^N. \tag{56}$$

We next consider two cases based on whether I_0 is empty. When I_0 is empty, we show that $\Delta LAG(\gamma, t, t_d) \leq \Delta lag(F, t, t_d)$. For the other case, we consider four subcases, and for each subcase, show that either $\Delta LAG(\gamma, t, t_d) \leq \Delta lag(F, t, t_d)$ or $\Delta LAG(\gamma, t, t + h_0 + b_0) \leq \Delta lag(F, t, t + h_0 + b_0)$ holds. If $\Delta LAG(\gamma, t, t_d) \leq \Delta lag(F, t, t_d)$ is shown to hold, then the lemma would be established for $u = t_d$. Otherwise, as shown in Fig. 11, no slot after H_0 is fully-idle. Hence, if I_0 is non-empty, and hence, v is fully-idle, then $v < t + h_0$ holds, and because B_0 is non-empty, $b_0 \geq 1$, and hence, $t + h_0 + b_0 > v + 1$ holds. Therefore, if $\Delta LAG(\gamma, t, t + h_0 + b_0) \leq \Delta lag(F, t, t + h_0 + b_0)$ is shown to hold when I_0 is non-empty, the lemma would be established for $u = t + h_0 + b_0$.

Case 1: I_0 is empty. Because I_0 is empty, $b_0 = 0$ and $L_0 = L_0^N = 0$. If $W_{\max} \leq f$ holds, then from (53) and (56), and $\Delta_f > 0$, $\Delta LAG(\gamma, t, t_d) < \Delta lag(F, t, t_d)$ follows. Hence, in the rest of the proof, we assume $W_{\max} > f$. Therefore, by (53), $\Delta LAG(\gamma, t, t_d) \leq L \cdot f + L^N \cdot (W_{\max} - f - 1)$, and by (56), $\Delta lag(F, t, t_d) = L(f + \Delta_f) - L^N$. By Lemma 4(c),

$$\begin{array}{ll} & LAG(\gamma, t_d) = 1 \quad \Rightarrow \quad \Delta LAG(\gamma, t, t_d) + LAG(\gamma, t) = 1 \quad \Rightarrow \quad L \cdot f + L^N \cdot (W_{\max} - f - 1) + LAG(\gamma, t) \geq 1 \\ \Rightarrow & L \cdot f + L^N \cdot (W_{\max} - f - 1) + 1 > 1 \quad \{\text{from the statement of the lemma, 38), and (24), } LAG(\gamma, t) < 1\} \\ \Rightarrow & L > \dfrac{L^N(1 + f - W_{\max})}{f} \end{array} \tag{57}$$

Because $W_{\max} > f$, by (8) and (23), $\Delta_f \geq (\frac{W_{\max} - f}{1 + f - W_{\max}}) \cdot f$ holds. Hence, by (57), $L \cdot \Delta_f > L^N(W_{\max} - f)$ holds. Therefore, using expressions derived above (for ΔLAG and Δlag), $\Delta LAG(\gamma, t, t_d) - \Delta lag(F, t, t_d) \leq L^N(W_{\max} - f) - L \cdot \Delta_f < 0$ follows, establishing the lemma.

Case 2: I_0 is nonempty. To prove the lemma for this case, we first show (59) and (60) below. To show (59), we show that $lag(F, t + h_0) = h_0(f + \Delta_f) - h_0^N$. Because I_0 is nonempty, by (39) and (38), we have the following.

(V) v is fully-idle, $t \leq v$, and $lag(F, t) = LAG(\gamma, t)$.

Since v is a fully-idle slot, by Lemma 10, every slot in $[0, v + 1)$ is fully-idle. Therefore, no task in γ is active in $[0, v + 1)$, and hence, $ideal(\gamma, 0, t') = 0$, for all $t' \leq v + 1$, and we have the following.

$$(\forall t' : 0 \leq t' \leq v + 1 :: LAG(\gamma, t') = 0) \tag{58}$$

(V) and (58) imply $lag(F, t) = 0$. Hence, by (15), $lag(F, t + h_0) = lag(F, t) + ideal(F, t, t + h_0) - actual(F, t, t + h_0) = ideal(F, t, t + h_0) - actual(F, t, t + h_0)$. Because F is allocated $f + \Delta_f$ time in every slot in an ideal schedule, and is allocated in every non-tight slot only in an actual schedule, $lag(F, t + h_0) = h_0(f + \Delta_f) - h_0^N$. By (24), $lag(F, t + h_0) > -1$, and hence, we have

$$h_0^N - h_0(f + \Delta_f) < 1. \tag{59}$$

By Lemma 4(c), $LAG(\gamma, t_d) = 1$ and hence, by(53), we have $\Delta LAG(\gamma, t, t_d) = (b_0 + L)f + L^N(W_{\max} - f - 1) = 1$, which implies

$$b_0 + L = \frac{1 - L^N(W_{\max} - f - 1)}{f}. \tag{60}$$

By (53) and (56), if $W_{\max} > f$, we have

$$\begin{array}{rcl} \Delta LAG(\gamma, t, t_d) - \Delta lag(F, t, t_d) & \leq & (b_0 + L)f - b_0^N + L^N(W_{\max} - f - 1) - (L + h_0 + b_0)(f + \Delta_f) + L^N + b_0^N + h_0^N \\ & & \{\text{after substituting } L_0 = h_0 + b_0 \text{ and } L_0^N = h_0^N + b_0^N\} \\ & = & L^N(W_{\max} - f) - (L + b_0)\Delta_f - h_0(f + \Delta_f) + h_0^N \\ & < & L^N(W_{\max} - f) - (L + b_0)\Delta_f + 1 \qquad \{\text{by (59)}\} \\ & \leq & L^N(W_{\max} - f) - \left(\dfrac{1 - L^N(W_{\max} - f - 1)}{f}\right)\Delta_f + 1 \quad \{\text{by (60)}\}. \end{array} \tag{61}$$

Similarly, if $W_{\max} \leq f$, by (53) and (56), we have

$$\Delta LAG(\gamma, t, t_d) - \Delta lag(F, t, t_d) \quad < \quad -(L + b_0)\Delta_f + 1 \tag{62}$$

$$\leq \quad -\left(\frac{1 - L^N(W_{\max} - f - 1)}{f}\right)\Delta_f + 1 \qquad \{\text{by (60)}\}.$$

We now consider the following subcases.

Subcase 2(a): $W_{\max} \geq f + 1/2$. For this subcase, by (8), $\Delta_f = \frac{W_{\max}-f}{1+f-W_{\max}} \times f$. Therefore, by (61),

$$\begin{aligned}\Delta LAG(\gamma, t, t_d) - \Delta lag(F, t, t_d) \quad &< \quad L^N(W_{\max} - f) - \left(\frac{1 - L^N(W_{\max} - f - 1)}{f}\right)\left(\frac{W_{\max} - f}{1 + f - W_{\max}}\right) f + 1 \\ &= \quad 1 - \frac{W_{\max} - f}{1 + f - W_{\max}} \quad \leq \quad 1 - \frac{f + 1/2 - f}{1 + f - f - 1/2} \quad = \quad 0.\end{aligned}$$

Subcase 2(b): $f < W_{\max} < f + 1/2$ and $\min(f, \frac{1}{\omega-1}) = f$. For this case, by (8) and (23), we have $\Delta_f \geq f$ and so, by (61), $\Delta LAG(\gamma, t, t_d) - \Delta lag(F, t, t_d) < L^N(W_{\max} - f) - \left(\frac{1-L^N(W_{\max}-f-1)}{f}\right) f + 1 = L^N(W_{\max} - f) - (1 - L^N(W_{\max} - f - 1)) + 1 = L^N(W_{\max} - f) + L^N(W_{\max} - f - 1) = 2L^N(W_{\max} - f) - L^N$ holds, which by $W_{\max} < f + 1/2$, implies that $\Delta LAG(\gamma, t, t_d) - \Delta lag(F, t, t_d) < 0$.

Subcase 2(c): $W_{\max} \leq f$. For this case, by (8) and (23), we have $\Delta_f = \frac{1}{\omega}$. Also, $t + h_0 - 1$ is the last fully-idle slot in $\mathcal{S}_\gamma$. Hence, by Lemma 11, $t_d \geq t + h_0 + \omega$, *i.e.*, $t_d - (t + h_0) = b_0 + L \geq \omega$ holds. So, by (62), $\Delta LAG(\gamma, t, t_d) - \Delta lag(F, t, t_d) < -\omega(1/\omega) + 1 = 0$.

Subcase 2(d): $f < W_{\max} < f + 1/2$ and $\min(f, \frac{1}{\omega-1}) = \frac{1}{\omega-1}$. By (8) and (23), we now have $\Delta_f = \frac{1}{\omega-1}$. Because $t + h_0 - 1$ is the last fully-idle slot in $\mathcal{S}_\gamma$, by Lemma 11, there are no holes in $[t + h_0, t + h_0 + \omega)$, hence $b_0 \geq \omega - 1$ holds. By (18), $LAG(\gamma, t + h_0 + b_0) = LAG(\gamma, t) + ideal(\gamma, t, t + h_0 + b_0) - actual(\gamma, t, t + h_0 + b_0)$. Because H_0 is fully idle, no task of γ is active there. Hence, $ideal(\gamma, t, t + h_0 + b_0) \leq b_0 \cdot (I + f)$. Because B_0 is busy, I (resp., $I + 1$) tasks are scheduled in every tight (resp., non-tight) slot. Hence, by (58), $LAG(\gamma, t + h_0 + b_0) \leq b_0 \cdot (I + f) - b_0 \cdot I - b_0^N = b_0 \cdot f - b_0^N$. Similarly, $lag(F, t + h_0 + b_0) = lag(F, t) + ideal(F, t, t + h_0 + b_0) - actual(F, t, t + h_0 + b_0)$. Because $lag(F, t) = 0$, we have $lag(F, t + h_0 + b_0) = (b_0 + h_0) \cdot (f + \Delta_f) - b_0^N - h_0^N$, which by (59) implies $lag(F, t + h_0 + b_0) \geq -1 + b_0 \cdot (f + \Delta_f) - b_0^N$. Because $\Delta_f \geq \frac{1}{\omega-1}$ and $b_0 \geq \omega - 1$,we have $lag(F, t + h_0 + b_0) \geq -1 + (\omega - 1) \cdot (f + \frac{1}{\omega-1}) - b_0^N \geq b_0 \cdot f - b_0^N$. Thus, $LAG(\gamma, t + h_0 + b_0) \leq lag(\gamma, t + h_0 + b_0)$. ■

By (28), there exists a u, where $0 \leq u < t_d$, such that $LAG(\tau, u) \leq lag(F, u)$ and $LAG(\tau, u+1) > lag(F, u+1)$. Let t be the largest such u. Then, by Lemma 12, there exists a $t' \leq t_d$ such that $LAG(\tau, t') \leq lag(F, t')$. If $t' = t_d$, then (27) is contradicted, and if $t' < t_d$, then (27) contradicts the maximality of t. Thus, our assumptions in Defs. 3 and 4 are incorrect and Δ_f given by (8) is a sufficient inflation factor to avoid deadline misses. Theorem 1 follows. (This result can be extended to apply when “early” subtask releases are allowed, as defined in [4], at the expense of a slightly more complicated proof.)