Real-Time Scheduling on Multicore Platforms *

James H. Anderson, John M. Calandrino, and UmaMaheswari C. Devi
Department of Computer Science
The University of North Carolina at Chapel Hill

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.*

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.

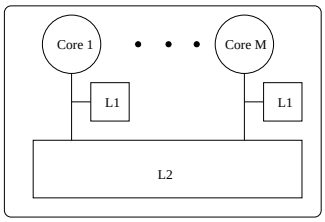

Figure 1: Multicore architecture.

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 only 10-30 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.

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

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

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 a four-core system 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 of weight (*i.e.*, utilization) 0.6 and with a working-set size of 200 KB (Group A), and four tasks of weight 0.3 and with a working-set size of 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 [6, 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, pipeline flushes, *etc.*, which (in comparison to L2 misses) 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.

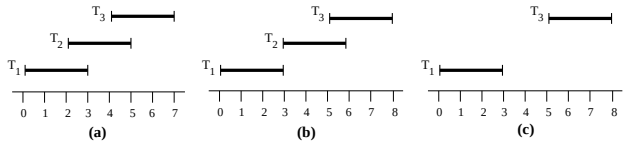

Figure 2: **(a)** Windows of subtasks $T_1, \ldots, T_3$ of a periodic task T of weight $3/7$. **(b)** T as an IS task; T_2 is released one time unit late. **(c)** T as a GIS task; T_2 is absent and T_3 is released one time unit late.

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 [6, 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 a 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.

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

In (1), $\theta(T_i)$ denotes the *offset* of T_i. The offsets of T's various subtasks are nonnegative and satisfy the following: $k > i \Rightarrow \theta(T_k) \geq \theta(T_i)$. 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 follows from (1).

Lemma 1 (from [5]) *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}, ...,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.

Scheduling algorithms. Pfair scheduling algorithms function by scheduling subtasks on an earliest-deadline-first basis. Tie-breaking rules are used in case two subtasks have the same deadline. The most efficient optimal algorithm known is PD^2 [5, 22], which uses two tie-breaks. PD^2 is *optimal*, *i.e.*, it correctly schedules any GIS task system τ for which $\sum_{T \in \tau} wt(T) \leq M$ holds.

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 $W_{\max}$.)

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

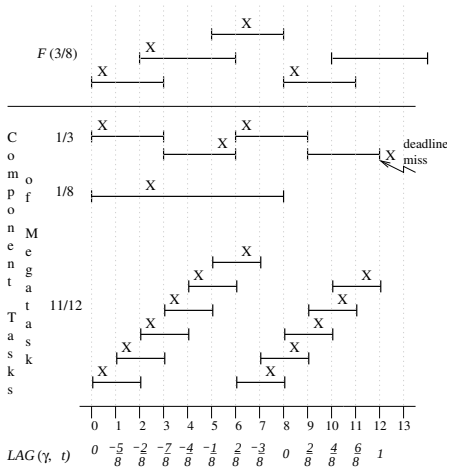

Figure 3: PD^2 schedule for the component (GIS) tasks of a megatask γ with $W_{sum} = 1 + \frac{3}{8}$. F represents the fictitious task associated with γ. γ is scheduled using its ideal weight by a top-level PD^2 scheduler. The slot in which a subtask is scheduled is indicated using an "X." γ is allocated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

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 previously proposed 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.

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.)

$$W_{sum} = \sum_{T \in \gamma} wt(T) = I + f \quad (2)$$

$$W_{\max} = \max_{T \in \gamma} wt(T) \quad (3)$$

$$\omega_{\max} = \lceil 1/W_{\max} \rceil \quad (4)$$

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.)

$$\omega = \begin{cases} \min(\text{smallest window length of task of rank} \\ (\omega_{\max} \cdot I + 1), 2\omega_{\max}), & \text{if } W_{\max} = \frac{1}{k},\ k \in \mathbb{N}^+ \\ \min(\text{smallest window length of task of rank} \\ ((\omega_{\max} - 1) \cdot I + 1), 2\omega_{\max} - 1), & \text{otherwise} \end{cases} \quad (5)$$

Then, a scheduling weight W_{sch} for γ may be computed using (6), where Δ_f is given by (7).

$$W_{sch} = W_{sum} + \Delta_f \quad (6)$$

$$\Delta_f = \begin{cases} \left(\frac{W_{\max} - f}{1 + f - W_{\max}}\right) \times f, & \text{if } W_{\max} \geq f + 1/2 \\ \min(1 - f, \max(\left(\frac{W_{\max} - f}{1 + f - W_{\max}}\right) \times f, \\ \quad \min(f, \frac{1}{\omega - 1}))), & \text{if } f + 1/2 > W_{\max} > f \\ \min(1 - f, \frac{1}{\omega}), & \text{if } W_{\max} \leq f \\ 0, & \text{if } f = 0 \end{cases} \quad (7)$$

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 $I = 1$ and $f = \frac{11}{20}$. Since $W_{\max} < f$, by (7), $\Delta_f = \min(1 - f, \frac{1}{\omega})$. We determine ω as follows. By (4), $\omega_{\max} = 3$. 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 (6), 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 (6), 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* (6)*, 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 wasteful. The various cases in (7) 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 real-time 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. (Later, in Sec. 4.2, we comment on why these cache sizes were chosen.) 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.

Name	No. Tasks	Task Properties	No. Cores	No. 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.

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 & 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.

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.

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.

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 WSSs 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 Niagara 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 tabulated 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 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

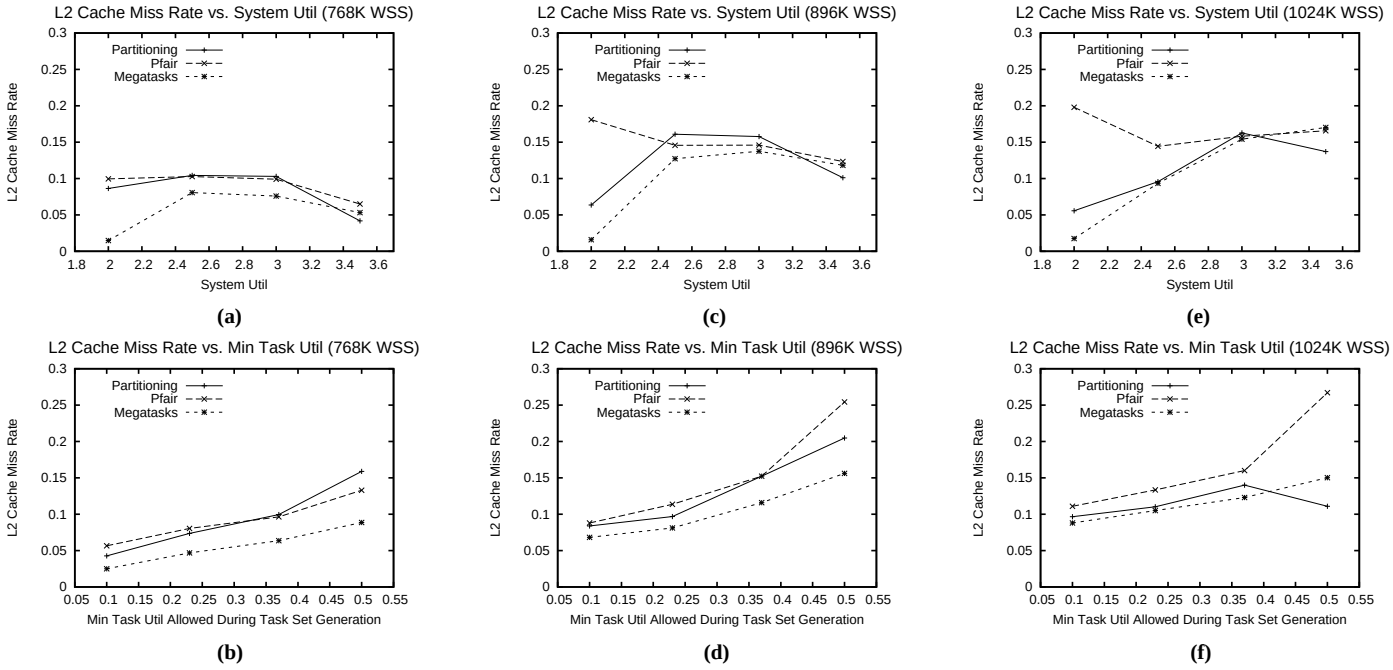

Figure 4: L2 cache miss rate versus both total system utilization (top) and minimum task utilization (bottom). The different columns correspond (left to right) to total WSSs of 1.5, 1.75, and 2.0 times the L2 cache capacity, respectively.

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 at least an order 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."

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.

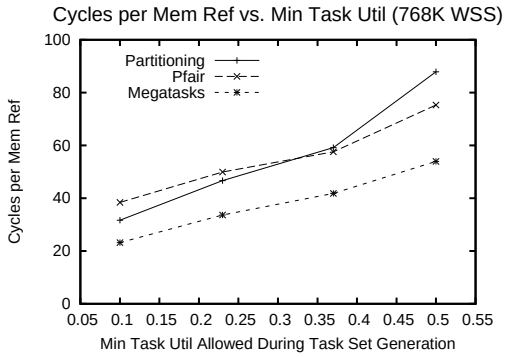

Figure 5: Cycles-per-memory-reference for the data in Fig. 4(b).

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 partitioning 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

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 partitioning counts is included in the Pfair (megatasking) counts.

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).

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, J. Calandrino, and U. Devi. Real-time scheduling on multicore platforms (full version). http://www.cs.unc.edu/∼anderson/papers.

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

[4] 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.

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

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

[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.

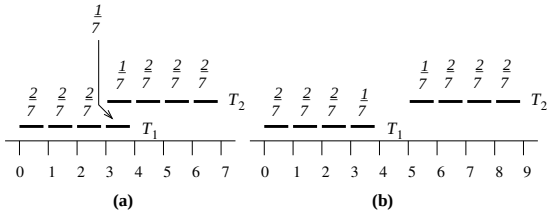

Figure 6: Allocation in an ideal fluid schedule for the first two subtasks of a task T of weight $2/7$. The share of each subtask in each slot of its window ($f(T_i, u)$) is marked. In **(a)**, no subtask is released late; in **(b)**, T_2 is released late. $share(T, 3)$ is either $2/7$ or $1/7$ depending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

[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 [3, 4, 5, 6, 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 $ideal(T, t_1, t_2)$ denote the processor share (or allocation) that T receives in an ideal fluid schedule in $[t_1, t_2)$. $ideal(T, t_1, t_2)$ is defined in terms of $share(T, u)$, which is the share (or fraction) of slot u assigned to task T. $share(T, u)$ is defined in terms of a similar *per-subtask* function f:

$$f(T_i, u) = \begin{cases} (\left\lfloor \frac{i-1}{wt(T)} \right\rfloor + 1) \times wt(T) - (i-1), & u = r(T_i) \\ i - (\left\lceil \frac{i}{wt(T)} \right\rceil - 1) \times wt(T), & u = d(T_i) - 1 \\ wt(T), & r(T_i) < u < d(T_i) - 1 \\ 0, & \text{otherwise.} \end{cases} \quad (8)$$

Using (8), it follows that $f(T_i, u)$ is at most $wt(T)$. Given f, $share(T, u)$ can be defined as $share(T, u) = \sum_i f(T_i, u)$, and then $ideal(T, t_1, t_2)$ as $\sum_{u=t_1}^{t_2-1} share(T, u)$. The following is proved in [22] (see Fig. 6).

$$(\forall u \geq 0 :: share(T, u) \leq wt(T)) \quad (9)$$

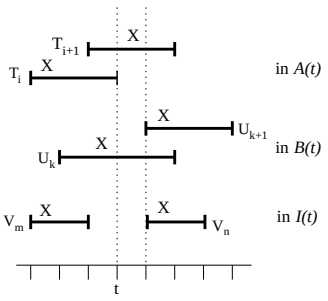

Figure 7: Classification of three GIS tasks T, U, and V at time t. The slot in which each subtask is scheduled is indicated by an "X."

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

$$\begin{aligned} lag(T, t, \mathcal{S}) &= ideal(T, 0, t) - actual(T, 0, t, \mathcal{S}) \\ &= \sum_{u=0}^{t-1} share(T, u) - \sum_{u=0}^{t-1} \mathcal{S}(T, u). \end{aligned} \quad (10)$$

(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

$$(\forall t, T \in \tau :: -1 < lag(T, t) < 1). \quad (11)$$

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 (11). Letting $0 \leq t' \leq t$, from (10), we have

$$lag(T, t+1) = lag(T, t) + share(T, t) - \mathcal{S}(T, t), \quad (12)$$

$$\begin{aligned} lag(T, t+1) &= lag(T, t') + ideal(T, t', t+1) - \\ &\quad actual(T, t', t+1). \end{aligned} \quad (13)$$

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

$$LAG(\tau, t) = \sum_{T \in \tau} lag(T, t). \quad (14)$$

Letting $0 \leq t' \leq t$, from (12)–(14), we have

$$\begin{aligned} LAG(\tau, t+1) &= LAG(\tau, t) + \\ &\quad \sum_{T \in \tau} (share(T, t) - \mathcal{S}(T, t)), \end{aligned} \quad (15)$$

$$\begin{aligned} LAG(\tau, t+1) &= LAG(\tau, t') + \\ &\quad ideal(\tau, t', t+1) - actual(\tau, t', t+1). \end{aligned} \quad (16)$$

Task classification. A GIS task U is *active* at time t if it has a subtask U_j such that $r(U_j) \leq t < d(U_j)$. The set $A(t)$ ($B(t)$) includes all active tasks scheduled (not scheduled) at t. The set $I(t)$ includes all tasks that are inactive at t. (See Fig. 7.)

Proof of Theorem 1

We now prove that W_{sch}, given by (6), is a sufficient scheduling weight. It can be verified that W_{sch} 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 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

$$f > 0 \ \wedge \ \Delta_f < 1 - f. \tag{17}$$

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 (11), the following holds. (We assume that the total number of processors is at least the total weight of all the megatasks after reweighting and any free tasks.)

$$(\forall t :: -1 < lag(F, t, \mathcal{S}) < 1) \tag{18}$$

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

Defn. 1: t_d is the earliest time that the component task system of any megatask misses a deadline under PD^2, when the megatask itself is scheduled by the root-level PD^2 scheduler according to its scheduling weight. ■

Defn. 2: 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. ■

As noted earlier, the setup here is similar to that used by Srinivasan and Anderson in the optimality proof of PD^2 [22], except that the number of processors allocated to a megatask is not constant. Despite this difference, a number of properties proved in [22] apply here, so we borrow them without proof. 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 $\mathcal{S}_\gamma$ (as defined earlier in (T1)) at any time t is denoted $LAG(\gamma, t, \mathcal{S}_\gamma)$ and is given by

$$LAG(\gamma, t, \mathcal{S}_\gamma) = \textstyle\sum_{T \in \gamma} lag(T, t, \mathcal{S}_\gamma). \tag{19}$$

By (9),

$$\begin{aligned} share(\gamma, t, \mathcal{S}_\gamma) &= \textstyle\sum_{T \in \gamma} share(T, t, \mathcal{S}_\gamma) \\ &\leq \textstyle\sum_{T \in \gamma} wt(T) \ = \ I + f. \end{aligned} \tag{20}$$

By (6), the 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. Slot t is a non-tight iff F is allocated in $\mathcal{S}$. In Fig. 3, slots 0 and 2 are non-tight, whereas slot 1 is tight.

Holes. If k of the processors assigned to γ in slot t are idle, then we say that there are k *holes* in $\mathcal{S}_\gamma$ at t.

Defn. 3: A slot in which every processor allocated to γ is idle (busy) is called a *fully-idle slot* (*busy slot*) for γ. A slot 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 2 (from [22]) *The properties below 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); (c) follows from (b). Part (d) holds because the subtask missing its deadline could otherwise be scheduled at $t_d - 1$. By Lemma 2(c) and (18),

$$LAG(\gamma, t_d, \mathcal{S}_\gamma) > lag(F, t_d, \mathcal{S}). \tag{21}$$

Because $LAG(\gamma, 0, \mathcal{S}_\gamma) = lag(F, 0, \mathcal{S}) = 0$, by (21),*

$$\begin{aligned} (\exists u : u < t_d :: LAG(\gamma, u) &\leq lag(F, u) \ \wedge \\ LAG(\gamma, u + 1) &> lag(F, u + 1)). \end{aligned} \tag{22}$$

In the remainder of the proof, we show that for every u as defined in (22), there exists 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 2(c), and hence, to our assumption that γ misses a deadline at t_d.

The next lemma shows that the lag inequality $LAG(\gamma, t) \leq lag(F, t)$ can be violated across slot t only if there are holes in t. 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 work on Pfair scheduling [22].

Lemma 3 *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.*

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. 8. As shown in this figure, if task T is in $B(t)$ (as defined earlier in this appendix), then no subtask of T with release time prior to t can have its deadline later than $t + 1$. Otherwise, because there is a hole in every slot in $[t + 1, u)$, removing such a subtask would not cause 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

*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.

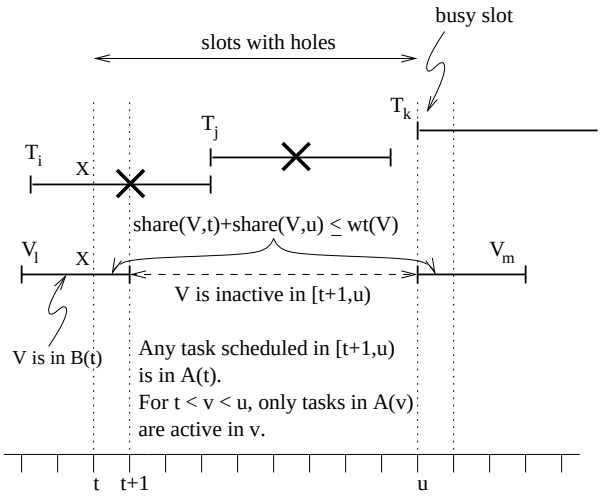

Figure 8: Lemma 4. The slot in which a subtask is scheduled is indicated with an "X." If T is in $B(t)$, subtasks like T_i or T_j cannot exist. Also, a task in $B(t)$ is inactive in $[t+1, u)$.

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. A full proof is available in [2].

Lemma 4 *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., *$t + 1 \leq u < t_d$) in $\mathcal{S}_\gamma$. Then, $ideal(\gamma, t, u + 1) = \sum_{s=t}^{u} \sum_{T \in \gamma} share(T, s) \leq I + f + \sum_{s=t}^{u-1} |A(s)|W_{\max}$.*

The next lemma concerns fully-idle slots.

Lemma 5 *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 left displacement occurs, then the displaced 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). ■

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

Lemma 6 *Let $t < t_d$ be a slot such that $LAG(\gamma, t) \leq lag(F, t)$, but $LAG(\gamma, t + 1) > lag(F, t + 1)$. Then, there exists a time u, where $t + 1 < u \leq t_d$, such that $LAG(\gamma, u) \leq lag(F, u)$.*

Proof: Let $\Delta LAG(\gamma, t_1, t_2) = LAG(\gamma, t_2) - LAG(\gamma, t_1)$, where $t_1 < t_2$, and let $\Delta lag(F, t_1, t_2)$ be analogously defined. It suffices to show that $\Delta LAG(\gamma, t, u) \leq \Delta lag(F, t, u)$, where u is as defined in the statement of the lemma.

By the statement of the lemma and Lemma 3, there is at least one hole in t, and hence, t is either fully- or partially-idle. These two cases differ somewhat, and due to space constraints, we present only the case where t is partially-idle. A complete analysis is available in [2]. (In the case of multiprocessors, a fully-idle slot provides a clean starting point for the analysis, and hence, in a sense, the partially-idle case is more interesting.) Thus, in the rest of the proof, assume that t is partially-idle. By this assumption and Lemma 5, we have the following.

(I) No slot in $[t, t_d)$ is fully-idle.

Because t is partially-idle, by Lemma 2(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. 9, where each subinterval is either partially-idle or busy. Because t is partially-idle, the first subinterval is partially-idle. Similarly, because there is no hole in $t_d - 1$, the last subinterval is busy. By (I), no slot in $[t, t_d)$ is fully-idle. Therefore, the intermediate subintervals of $\mathcal{I}$ alternate between busy and partially-idle, in that order. In the rest of the proof, the notation in Fig. 10 will be used, where the subintervals H_k, B_k, and I_k, where $1 \leq k \leq n$, are as depicted in Fig. 9.

In order to show that there exists a u, where $t + 1 < u \leq t_d$ and $\Delta LAG(\gamma, t, u) \leq \Delta lag(F, t, u)$, we compute the ideal and actual allocations to the tasks in γ and to F in $\mathcal{I}$. By Lemma 4, the total allocation to the tasks in γ in H_k and the first slot of B_k in the ideal schedule is given by $ideal(\gamma, t^s_{H_k}, t^s_{B_k} + 1) \leq I + f + \sum_{i=1}^{h_k} |A(t + P_{k-1} + i - 1)| \cdot W_{\max}$. By (20), the tasks in γ are allocated at most $W_{sum} = I + f$ time in each slot in the ideal schedule. Hence, the total ideal allocation to the tasks in γ in 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 to the tasks in γ in $\mathcal{I}$ is given by

$$ideal(\gamma, t, t_d) \leq \sum_{k=1}^{n} \left(\left(\sum_{i=1}^{h_k} (|A(t+P_{k-1}+i-1)| \cdot W_{\max}) \right) + b_k \cdot (I+f) \right). \quad (23)$$

The number of processors executing tasks of γ in $\mathcal{S}_\gamma$ is $|A(t')|$ for a slot t' with a hole, and is I (resp., $I + 1$) for a busy tight (resp., non-tight) slot. Hence,

$$actual(\gamma, t, t_d) = \sum_{k=1}^{n} \left(\left(\sum_{i=1}^{h_k} |A(t + P_{k-1} + i - 1)| \right) + I \cdot b_k^{\mathrm{T}} + (I + 1) \cdot (b_k - b_k^{\mathrm{T}}) \right). \quad (24)$$

By (23) and (24), we have

$$\begin{aligned} \Delta LAG(\gamma, t, t_d) &= LAG(\gamma, t_d) - LAG(\gamma, t) \\ &= ideal(\gamma, t, t_d) - actual(\gamma, t, t_d) \quad \{\text{by (16)}\} \\ &\leq \sum_{k=1}^{n} \left(\left(\sum_{i=1}^{h_k} (|A(t+ P_{k-1}+i-1)| \cdot (W_{\max} - 1)) \right) + b_k^{\mathrm{T}} \cdot f + (b_k - b_k^{\mathrm{T}})(f-1) \right) \quad (30) \\ &\leq \sum_{k=1}^{n} \left(\left(\sum_{i=1}^{h_k} (W_{\max} - 1) \right) + b_k^{\mathrm{T}} \cdot f + (b_k - b_k^{\mathrm{T}})(f-1) \right) \end{aligned}$$

$\{W_{\max} \leq 1$, and hence, (30) decreases with increasing $|A(t+P_{k-1}+i-1)|$. However, by (I), $H_1, \ldots, H_n$ are partially-idle, so, $|A(t+P_{k-1}+i-1)| \geq 1, 1 \leq k \leq n.\}$

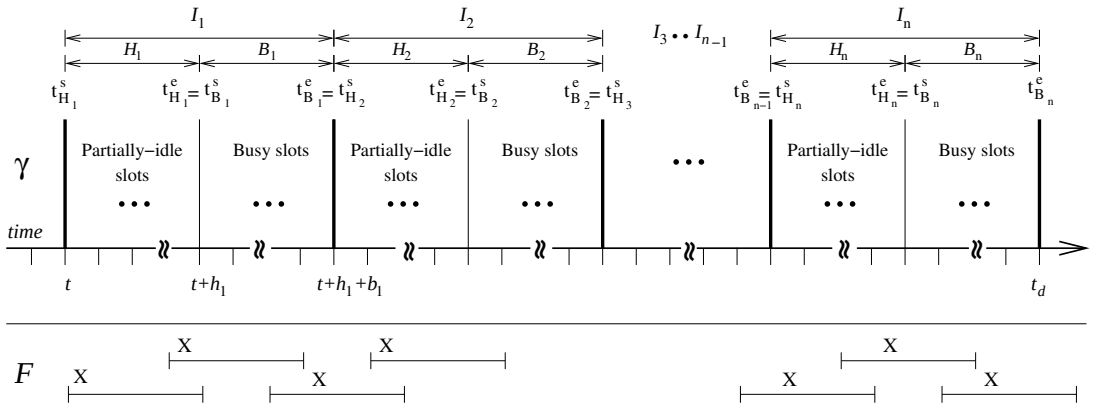

Figure 9: Subintervals of the interval $\mathcal{I} = [t, t_d)$ as explained in Lemma 6. Sample windows and allocations for the fictitious task corresponding to γ (after reweighting) are shown below the time line.

H_k	$\stackrel{\text{def}}{=}$	$[t^s_{H_k}, t^e_{H_k})$	$\{s =$ "start", $e =$ "end"$\}$
B_k	$\stackrel{\text{def}}{=}$	$[t^s_{B_k}, t^e_{B_k})$	
t	$\stackrel{\text{def}}{=}$	$t^s_{H_1}$	
t_d	$\stackrel{\text{def}}{=}$	$t^e_{B_n}$	
$t^e_{H_k}$	$\stackrel{\text{def}}{=}$	$t^s_{B_k}, 0 \le k \le n$	
$t^e_{B_k}$	$\stackrel{\text{def}}{=}$	$t^s_{H_{k+1}}, 0 \le k \le n-1$	
h_k	$\stackrel{\text{def}}{=}$	$t^e_{H_k} - t^s_{H_k}, 0 \le k \le n$	$\{= \lvert H(k)\rvert\}$
b_k	$\stackrel{\text{def}}{=}$	$t^e_{B_k} - t^s_{B_k}, 0 \le k \le n$	$\{= \lvert B(k)\rvert\}$
h^{T}_k (b^{T}_k)	$\stackrel{\text{def}}{=}$	no. of tight slots in H_k (B_k)	
h^{N}_k (b^{N}_k)	$\stackrel{\text{def}}{=}$	no. of non-tight slots in H_k (B_k)	
L	$\stackrel{\text{def}}{=}$	$\sum_{k=1}^{N}(h_k + b_k)$	(25)
L^{T}	$\stackrel{\text{def}}{=}$	$\sum_{k=1}^{N}(h^{\mathrm{T}}_k + b^{\mathrm{T}}_k)$	(26)
L^{N}	$\stackrel{\text{def}}{=}$	$\sum_{k=1}^{N}(h^{\mathrm{N}}_k + b^{\mathrm{N}}_k)$	(27)
P_k	$\stackrel{\text{def}}{=}$	$\sum_{i=1}^{k}(h_i + b_i)$	(28)
P_0	$\stackrel{\text{def}}{=}$	0	(29)

Figure 10: Notation for Lemma 6.

$$
\begin{aligned}
&= \textstyle\sum_{k=1}^{n}(h_k(W_{\max}-1)+b^{\mathrm{T}}_k\cdot f+(b_k-b^{\mathrm{T}}_k)(f-1)) \\
&= \textstyle\sum_{k=1}^{n}(h_k(W_{\max}-1)+b_k\cdot f-b_k+b^{\mathrm{T}}_k) \\
&= \textstyle\sum_{k=1}^{n}(h_k\cdot((W_{\max}-f-1)+f)+b_k\cdot f-b_k+b^{\mathrm{T}}_k) \\
&= \textstyle\sum_{k=1}^{n}((h_k+b_k)\cdot f + \\
&\quad h_k\cdot(W_{\max}-f-1)-b^{\mathrm{N}}_k) && \{b_k = b^{\mathrm{T}}_k + b^{\mathrm{N}}_k\} \\
&= L\cdot f+\textstyle\sum_{k=1}^{n}(h_k\cdot(W_{\max}-f-1)-b^{\mathrm{N}}_k) && \{\text{by (25)}\} \\
&= L\cdot f+\textstyle\sum_{k=1}^{n}(h^{\mathrm{T}}_k\cdot(W_{\max}-f-1)+ \\
&\quad h^{\mathrm{N}}_k\cdot(W_{\max}-f-1)-b^{\mathrm{N}}_k) && \{h_k = h^{\mathrm{T}}_k + h^{\mathrm{N}}_k\} \\
&\le L\cdot f+\textstyle\sum_{k=1}^{n}(h^{\mathrm{N}}_k\cdot(W_{\max}-f-1)-b^{\mathrm{N}}_k) && \{W_{\max}\le 1\} \\
&= L\cdot f-L^{\mathrm{N}}+\textstyle\sum_{k=1}^{n}h^{\mathrm{N}}_k\cdot(W_{\max}-f) && \{\text{by (27)}\} \\
&\le \begin{cases} L\cdot f+L^{\mathrm{N}}(W_{\max}-f-1), & W_{\max}>f \\ L\cdot f-L^{\mathrm{N}}, & W_{\max}\le f. \end{cases} && (31)
\end{aligned}
$$

We now determine the change in F's lag across $\mathcal{I}$. F receives an ideal allocation of $f + \Delta_f$ in every slot. Hence, by (25),

$$ideal(F,t,t_d) = \textstyle\sum_{k=1}^{n}(h_k + b_k)(f + \Delta_f) = L\cdot(f + \Delta_f). \quad (32)$$

In $\mathcal{S}$, F is allocated in every non-tight slot in $\mathcal{I}$. Hence, by (27),

$$actual(F,t,t_d) = \textstyle\sum_{k=1}^{n}(h^{\mathrm{N}}_k + b^{\mathrm{N}}_k) = L^{\mathrm{N}}. \quad (33)$$

Thus, by (13), the change in lag of F across $\mathcal{I}$ is given by

$$
\begin{aligned}
\Delta lag(F,t,t_d) &= lag(F,t_d) - lag(F,t) \\
&= ideal(F,t,t_d) - actual(F,t,t_d) \\
&= L\cdot(f+\Delta_f) - L^{\mathrm{N}}. \quad (34)
\end{aligned}
$$

We are now ready to show that $\Delta LAG(\gamma,t,t_d) \le \Delta lag(F,t,t_d)$, establishing the lemma with $u = t_d$.

If $W_{\max} \le f$ holds, then from (31), (34), and $\Delta_f > 0$, we have $\Delta LAG(\gamma,t,t_d) < \Delta lag(F,t,t_d)$. Hence, in the rest of the proof, we assume $W_{\max} > f$. In this case, by (31), $\Delta LAG(\gamma,t,t_d) \le L\cdot f + L^{\mathrm{N}}\cdot(W_{\max} - f - 1)$, and by (34), $\Delta lag(F,t,t_d) = L(f+\Delta_f) - L^{\mathrm{N}}$. By Lemma 2(c),

$$
\begin{aligned}
& LAG(\gamma,t_d) = 1 \\
\Rightarrow\ & \Delta LAG(\gamma,t,t_d) + LAG(\gamma,t) = 1 \\
\Rightarrow\ & L\cdot f + L^{\mathrm{N}}\cdot(W_{\max} - f - 1) + LAG(\gamma,t) \ge 1 \\
\Rightarrow\ & L\cdot f + L^{\mathrm{N}}\cdot(W_{\max} - f - 1) + 1 > 1 \\
& \{\text{from the statement of the lemma and (18), } LAG(\gamma,t) < 1\} \\
\Rightarrow\ & L > (L^{\mathrm{N}}(1 + f - W_{\max}))/f. \quad (35)
\end{aligned}
$$

Because $W_{\max} > f$, by (7) and (17), $\Delta_f \ge (\frac{W_{\max}-f}{1+f-W_{\max}})\cdot f$ holds. Hence, by (35), $L\cdot\Delta_f > L^{\mathrm{N}}(W_{\max} - f)$ holds. Therefore, using the expressions derived above for ΔLAG and Δlag, $\Delta LAG(\gamma,t,t_d) - \Delta lag(F,t,t_d) \le L^{\mathrm{N}}(W_{\max} - f) - L\cdot\Delta_f < 0$ follows, establishing the lemma. ■

Let t be the largest u satisfying (22). Then, by Lemma 6, there exists a $t' \le t_d$ such that $LAG(\tau,t') \le lag(F,t')$. If $t' = t_d$, then (21) is contradicted, and if $t' < t_d$, then (21) contradicts the maximality of t. Theorem 1 follows. (This result can be extended to apply when "early" subtask releases are allowed, as defined in [5], at the expense of a slightly more complicated proof.)