Parallel Real-Time Task Scheduling on Multicore Platforms [*]

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

Abstract

We propose a scheduling method for real-time systems implemented on multicore platforms that encourages individual threads of multithreaded *real-time tasks to be scheduled together. When such threads are cooperative and share a common working set, this method enables more effective use of on-chip shared caches.*

1 Introduction

Multicore architectures, which include several processors on a single chip, are being widely touted as a solution to the "thermal roadblock" imposed by single-core designs. Several chip makers have released dual-core chips, and a few designs with more than two cores have been announced as well. For instance, Sun recently released its eight-core Niagara chip, while Intel is expected to release chips with 80 cores within five years [6].

In most proposed multicore platforms, different cores share on-chip caches. To effectively exploit the available parallelism in these systems, such caches must not become performance bottlenecks. In fact, the issue of efficient cache usage on multicore platforms is one of the most important problems with which chip makers are currently grappling. 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 widely-studied multicore architecture shown in Fig. 1, where all cores are symmetric, single-threaded, and share an L2 cache.

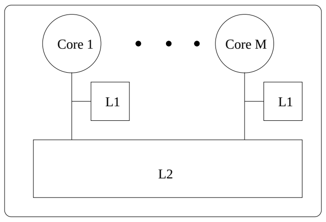


Figure 1: Multicore architecture.

In prior work pertaining to throughput-oriented systems, Fedorova *et al.* [7] noted that L2 misses affect performance to a much greater extent than L1 misses or pipeline conflicts. They showed that L2 contention can be reduced, and throughput improved, by discouraging threads that generate significant memory-to-L2 traffic from being co-scheduled. In recent work [1], we presented results comparable to those of Fedorova *et al.* but pertaining to *real-time* systems. Specifically, we showed that it is possible to discourage high-cache-impact tasks from being co-scheduled *while ensuring real-time constraints*.

The problem. In this paper, we consider the opposite issue of whether certain groups of real-time tasks can be *encouraged* to be co-scheduled. Specifically, we consider a periodic task model in which tasks may be *multithreaded*: a multithreaded task (MTT) consists of several (sequential) threads, which may have different execution costs but a common period. Our goal is to devise mechanisms that encourage a task's threads to be co-scheduled. Such encouragement would be beneficial in settings wherein a task's threads are cooperative and share a common working set. Specifically, when cooperating threads execute in close proximity in time, their accesses of common data will give rise to fewer L2 misses. Multithreaded tasks such as this arise naturally in many settings. For example, in multimedia applications, multiple threads may be useful for performing different functions on common data (*e.g.*, a frame of an MPEG video) at the same rate.

Related work. In work on parallel computing, it is well-known that the memory-reference patterns of threads can lead to co-scheduling choices that are either *constructive* or *destructive* [12]. However, to the best of our knowledge, we are the first to consider mechanisms for influencing such choices when analysis validating *real-time constraints* is required. Related work that lacks such analysis includes work on *symbiotic scheduling* [9, 11, 14] in (non-multicore) multithreaded systems. In symbiotic scheduling, the goal is to maximize the overall "symbiosis factor," which indicates how well various thread groupings perform when co-scheduled. Related work on multicore systems includes the work of Fedorova *et al.* [7] noted earlier, as well as a related paper [8] on a scheduling approach that encourages "fairness" in shared cache usage by different threads. In the latter paper, quality-of-service requirements are considered experimentally, but no real-time analysis is presented. Caching issues that arise when *non-real-time* multithreaded applications are implemented on multicore platforms have also been considered in prior work [4, 10].

Proposed approach. The essence of the problem at hand is to encourage *parallelism*: when one thread of an MTT is scheduled, *all* threads of that MTT should be scheduled. Unfortunately, perfect parallelism is not always possible. For example, scheduling in parallel two threads with an execution cost of 1.0 and a period of 2.0 on a two-processor sys-

[*]Work supported by a grant from Intel Corp. and by NSF grants CNS 0309825, CNS 0408996, and CCF 0541056.

tem that also includes a task with an execution cost of 3.0 and period of 4.0 will result in deadline misses. Additionally, we have shown that the general problem of optimizing for parallelism while respecting real-time constraints (not surprisingly) is NP-hard in the strong sense.

Due to these limitations, we have chosen to focus on minimizing a factor we call *spread*. For ease of explanation, we assume for now that each MTT is comprised of threads that have the same execution cost, and thus the same *utilization* or *weight* as well, as they share a common period. Later, this restriction will be removed. Given this assumption, if an MTT has a spread of k, then the i^{th} quantum of computation for each thread of the MTT must be scheduled within the interval $[t, t+k)$ for some t (treating each "time unit" as a quantum). Our goal, then, is to schedule MTTs so that real-time constraints are met and spread is minimized to the extent possible. Note that, though a spread of one is perfect parallelism, even with somewhat larger spreads, a potential for cache reuse exists. Our approach for minimizing spread while meeting real-time constraints is based upon three observations.

First, global scheduling algorithms, which use a single run queue, are more naturally suited to minimizing spread than partitioning approaches. This is particularly the case when using deadline-based scheduling methods. This is because "work" with a common deadline submitted at the same time will occupy consecutive slots in the scheduler's run queue, and thus, such work will be scheduled in close proximity over time, unless disrupted by later-arriving, higher-priority work. Based on this observation, we henceforth limit our attention to global, deadline-based scheduling approaches. Two such approaches will be considered in detail: the PD^2 Pfair scheduling algorithm [2] and the global earliest-deadline-first (EDF) algorithm. In Pfair scheduling, each "unit of work" is a quantum-length *subtask*, while under EDF, each unit is a *job* of arbitrary (but bounded) length.

Second, in all global, deadline-based scheduling methods known to us, the ability to meet timing constraints is not compromised if subtasks or jobs (as the case may be) are "early released," *i.e.*, allowed to become eligible for execution "early." This is depicted with respect to Pfair scheduling in Fig. 2. In Pfair scheduling, each subtask must be scheduled within a time window, the end of which is its deadline. Note that, in inset (b), allowing early releasing does not cause deadline misses.

Third, when a subtask or job is early released, it is *optional* as to whether the scheduler considers it for execution. We can exploit this fact to minimize disruptions to an MTT's threads caused by higher-priority work. As an example, consider the Pfair schedules in Fig. 3, where both tasks of weight $1/4$ are threads of the same MTT. Inset (a) shows a schedule without early releasing. In inset (b), all subtask windows are shifted right by one quantum, and all subtasks are early

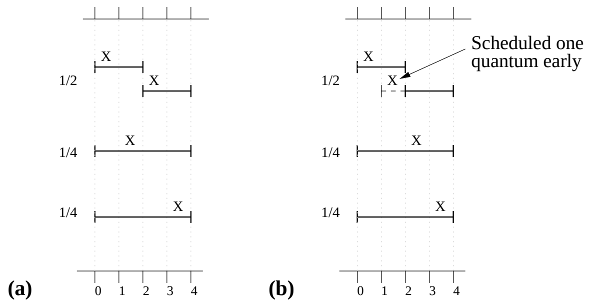


Figure 2: A one-processor Pfair schedule for a set of three tasks (of weight 1/2, 1/4, and 1/4, respectively) with **(a)** no early releasing and **(b)** early releases allowed by one quantum. Solid lines indicate subtask windows, dashed lines indicate where early releasing is allowed, and an "X" indicates when a subtask is scheduled.

released by one quantum, producing the same schedule as in (a). (All deadline comparisons are the same.) We refer to a schedule in which all subtask windows are right-shifted by k quanta and all subtasks are early released by k quanta as a k-*shifted schedule*. In inset (b), $k = 1$. The schedules in (a) and (b) both result in a spread of three for the MTT. In inset (c), we show that selectively allowing early releasing can reduce spread to two. Alternately, instead of shifting the schedule and early releasing subtasks, as in (b) and (c), we can instead consider a subtask to be optional for scheduling for the first k quanta after its release, and allow it to miss its deadline by up to k quanta, as shown in inset (d). Here, the dotted lines after each window indicate by how much each deadline could be missed (though no misses occur here). Note that there are "intermediate" cases between an unshifted and a k-shifted schedule. For example, if $k = 4$ is required by our method for a given task set, then we could choose instead to create a 3-shifted schedule, but consider a subtask optional for scheduling in the first quantum after its actual release. In this case, deadlines could be missed by at most one quantum. More generally, if subtasks can be early released to the extent we require, then no deadlines will be missed; otherwise, deadlines may be missed, but by bounded amounts only.

Contributions. Based upon the above observations, we have devised a set of rules for *guaranteeing* low spreads in deadline-based, global scheduling approaches. We present these rules by first focusing on PD^2 and by then explaining how to adapt them to EDF. In both cases, we consider first only MTTs with same-weight threads (as above) and then ease this restriction. In all cases, we establish the spread guarantees that are possible (though some properties are only stated or sketched, due to page limitations). As a final contribution, we evaluate the efficacy of the proposed rules in two ways. First, we assess the spread reductions they enable by presenting experiments involving randomly-generated task sets. Second, we assess the corresponding L2 miss-rate reductions that result by presenting experiments that involve running actual tasks on a multicore simulator. These exper-

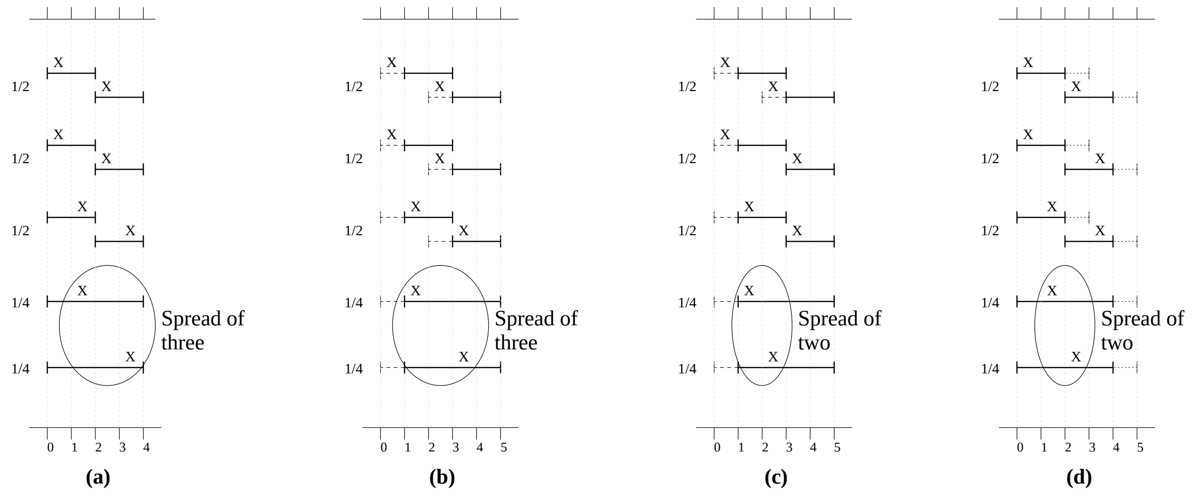


Figure 3: A two-processor Pfair schedule of a set of five tasks (three of weight 1/2, and two of weight 1/4) with **(a)** no early releasing; **(b)** early releasing by one quantum and all windows right-shifted by one quantum; **(c)** similarly-shifted windows, but selective early releasing; **(d)** no shifting or early releasing, but subtasks are considered optional within the first quantum after their release, and deadlines can be missed by one quantum.

iments show that our rules often reduce spreads to close to one (perfect parallelism) and can reduce L2 miss rates significantly.

The rest of this paper is organized as follows. In Sec. 2, we present a brief overview of Pfair and EDF scheduling. Then, in Sec. 3, we describe our spread-reduction approach and establish the PD^2 and EDF spread guarantees mentioned earlier. In Sec. 4, we present experimental results, and in Sec. 5, we conclude.

2 Background

In this section, we briefly introduce both Pfair [3, 15] and EDF scheduling. For simplicity, we consider only periodic task systems, though our results apply to sporadic and intra-sporadic [15] task systems as well. In a periodic task system τ, each task T releases successive *jobs* and is characterized by a per-job *execution cost* $T.e$ and a *period* $T.p$. Every $T.p$ time units, starting at time 0, T releases a new job with an execution cost of $T.e$ time units. Execution costs and periods are assumed to be integral. In Pfair scheduling, a time unit is actually a unit of processor allocation, and thus is referred to as a *quantum*. In EDF, a time unit is not necessarily a unit of allocation, but can be any convenient size such that execution costs and periods are integral. The quantity $T.e/T.p$ is the *weight*, or *utilization*, of T, denoted $wt(T)$.

Our goal is to schedule on M processors (or cores) a set of periodic tasks of total weight at most M, where some tasks correspond to threads within an MTT. For now, we assume each of an MTT's threads has the same execution cost (as well as period), but later this restriction is removed. We also assume the following.

(PM) Each MTT has at most M threads, the maximum parallelism achievable on M cores.

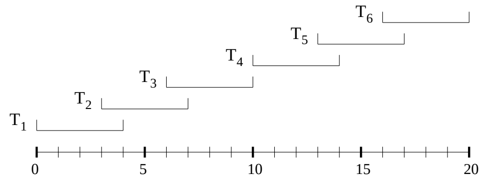


Figure 4: Windows for a task T of weight $3/10$. Three (six) subtasks have deadlines by time 10 (20), so if all deadlines are met, T's allocation up to these times matches its ideal allocation ($\frac{3}{10} \times 10$ and $\frac{3}{10} \times 20$).

2.1 Pfair Scheduling

Pfair scheduling algorithms [3, 15] allocate processor time one quantum at a time. The quantum-length time interval $[t, t+1)$, where $t \geq 0$, is called *slot* t. In each slot, each processor (task) can be assigned to at most one task (processor). Task migration is allowed. Per-quantum allocations are achieved by sub-dividing each task T into a sequence of quantum-length *subtasks*, denoted $T_1, T_2, \ldots$. Each subtask T_i has an associated *release* $r(T_i)$ and *deadline* $d(T_i)$, defined as follows.

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

The time-slot interval $[r(T_i), d(T_i))$ is called the *window* of T_i. Consecutive subtask windows of a task are either disjoint or overlap by one slot, as seen in Fig. 4. A task's windowing is defined to approximate an ideal (fluid) system that allocates $wt(T) \cdot L$ units of processor time to each task T in any interval of length L.

Pfair scheduling algorithms schedule tasks by scheduling their subtasks on an EDF basis. Tie-breaking rules are used when two subtasks have equal deadlines. The most efficient optimal algorithm known is PD^2 [2, 15], which uses two tie-breaking rules. Because it is optimal, PD^2 meets all subtask deadlines, as long as total utilization, $\sum_{T\in\tau} wt(T)$, does not exceed M. PD^2's ability to meet subtask deadlines is not affected if *early-release behavior* is allowed [2], *i.e.*, if subtasks can become eligible to execute before their windows. Note that, with early releasing, we can speak of a subtask being scheduled *before* its designated release time. The decision of whether to release a subtask early is arbitrary.

2.2 EDF Scheduling

Under (global) EDF scheduling, jobs are scheduled in order of increasing deadlines, with ties broken arbitrarily. EDF is *not* optimal, so tasks may miss their deadlines. It has been shown, however, that deadline tardiness under EDF is bounded [5, 16]. Additionally, jobs may be optionally and arbitrarily early released under EDF, as in Pfair scheduling, with no additional tardiness penalties. For EDF, we will denote the j^{th} job of a task T as T_j.

3 Spread-Cognizant Scheduling

In describing our method, we consider PD^2 and EDF separately.

3.1 Method Applied to PD^2

In general, we will use X to denote the spread guarantee we seek to establish. For PD^2, X is defined as follows, where $W_{\max} = \max_{T\in\tau} wt(T)$, and T can be any task (including a task corresponding to a thread of an MTT).

$$X = \begin{cases} 3, & \textbf{if } W_{\max} \leq 1/3 \\ 4, & \textbf{if } 1/3 < W_{\max} \leq 1/2 \\ 2 \times \lceil \frac{1}{1-W_{\max}} \rceil - 1, & \textbf{if } W_{\max} > 1/2 \end{cases} \quad (2)$$

We describe our method as additional rules for PD^2. We assume we are working with an $(X-1)$-shifted schedule (or alternately, that subtasks can miss their deadlines by up to $X-1$ quanta—see Fig. 3(d)). Three rules are required:

- **Urgent Tasks.** When subtask T_i is scheduled, where task T corresponds to a thread within some MTT R, and T is the first thread in R whose i^{th} subtask is scheduled, each subtask U_i, where U is also a thread of R and $U \neq T$, is flagged "urgent" until it is also scheduled.

- **Early-Release Eligibility.** A *non-urgent* subtask T_j at time t is "early-release eligible" at t if all of the following hold:

  (i) $r(T_j)-(X-1) \leq t < r(T_j)$ (*i.e.*, time t is within $X-1$ slots of the actual release time of subtask T_j).

  (ii) All subtasks T_k of T, where $k < j$, have already been scheduled prior to time t.

  (iii) $|\mathcal{U}| + |\mathcal{H}| < M$, where, at time t, $\mathcal{U}$ is the set of eligible urgent subtasks, and $\mathcal{H}$ is the set of non-urgent eligible subtasks T_k, where $r(T_k) \leq t$, such that each subtask in $\mathcal{H}$ has higher priority than at least one subtask in $\mathcal{U}$. Note that tasks in $\mathcal{H}$ are (by definition) not early-release eligible at time t.

  (iv) Subtask T_j is one of the $e = M - (|\mathcal{U}| + |\mathcal{H}|)$ highest-priority subtasks at time t satisfying (i) and (ii) above.

  A subtask T_j that is *urgent* at time t is "early-release eligible" at time t if conditions (i) and (ii) hold for it.

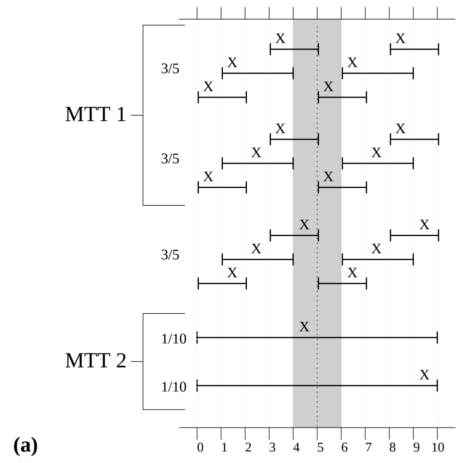
- **Priorities.** Eligible subtasks (early released or not) are scheduled using the same priority rules as in PD^2. In the case of a tie, urgent subtasks have higher priority, with the MTT identity used as a tie-break. (This ensures that MTTs achieve the lowest possible spread when nothing in the PD^2 priority rules would prevent it.)

These rules are illustrated in Fig. 5. For simplicity, we have assumed here that early releases occur by *one* slot instead of as implied by (2). With regular PD^2 (inset (a)), the task set achieves maximum spreads of two and six for MTTs 1 and 2, respectively. Our rules reduce the spread of MTT 2 to two (inset (b)), without changing the spread of MTT 1. This reduction happens because at time 5 in (b), the $1/10$-weight task not scheduled at that time in (a) is favored over the $3/5$-weight tasks by the *Urgent Tasks* rule. Note that the *Early-Release Eligibility* rule only allows one of the $3/5$-weight subtasks released at time 6 to become early-release eligible at time 5. Note also that, if the task set included some additional tasks of weight $1/10$ that were eligible at time 5, the *Priorities* rule would ensure that the urgent subtask was scheduled first, and MTT 2 would still have a spread of two.

In an appendix, we prove the following.

Theorem 1 *If PD^2 is modified as described above, subtasks are early-release eligible $X-1$ quanta before their actual release times, and all task threads in the same MTT have the same weight, then the spread of any MTT is no greater than X as defined in* (2).

Removing the "same-weight" restriction. Define two subtasks as being *equivalent* if they have equal releases and deadlines (as defined by (1)) and PD^2 tie-breaks. Then, the essence of the rules presented above is that, by exploiting early-release behavior, equivalent subtasks can be made to execute within a constrained time interval. If the threads of an MTT are all of the same weight, then their i^{th} subtasks are



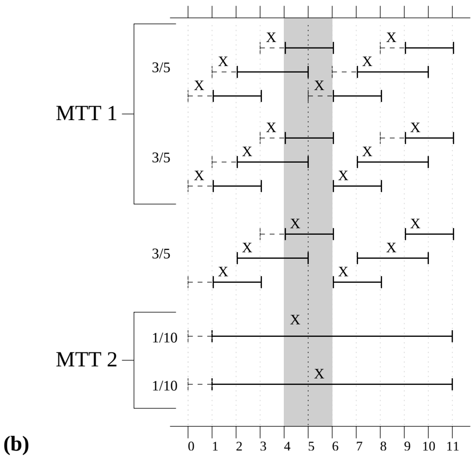


Figure 5: An example two-processor schedule with **(a)** regular PD^2, and **(b)** PD^2 using our method.

all equivalent. However, if the threads of an MTT are of different weights, then this will no longer be the case. Nonetheless, many subtasks will share a common deadline and tie breaks—in particular, this will definitely be the case for the final subtask per job of each thread, and will likely be the case for some other subtasks as well. Such subtasks can be made to be "equivalent" as before, by defining their releases to be the same. Here, again, we can exploit early-release behavior, specifically, by defining the release of each subtask in a job of a thread to be the same as the first such subtask (though subtasks must still execute in sequence); the first subtask per job would be early released by $X - 1$ quanta, as before. With this change, we can offer the same spread guarantees as earlier, but we must alter the definition of spread slightly to account for the fact that two equivalent subtasks may have indices that differ significantly. The condition for an MTT to have a spread of k is now as follows: if a subtask T_i of some thread T of the MTT is scheduled at time t, then for each thread U of the MTT with an unscheduled subtask U_j that is equivalent to T_i, either subtask U_j is scheduled at time t, or some subtask U_m is scheduled in the interval $[t + 1, t + k)$, where $m \leq j$. (Note that if some U_m, where $m < j$, is scheduled in $[t + 1, t + k)$, then we still have a potential for cache reuse.) With this new spread definition, and a few cosmetic changes to the rules stated earlier, Theorem 1 can be strengthened to remove the "same-weight" restriction.

We illustrate some of the nuances of this new spread definition by considering an example, shown in Fig. 6. In this example, all tasks with period 10 are threads in the same MTT. We define the (early) release time of each subtask in a job to be the same, with the first being early released by one quantum. At time 1, subtasks T_1, U_1, and V_1 are scheduled; however, by our new definition of spread, this does *not* result in a spread of one. While U_1 and V_1 are clearly equivalent, T_1 is equivalent to neither subtask (because its deadline is at time 5, while U_1 and V_1 have a deadline at time 11). However, T_3 (not shown) *is* equivalent, as it has the same deadline and tie-breaks as U_1 and V_1. Therefore, when T_2 is scheduled at time 2, the condition for a spread of two is met, as T_2 both precedes T_3 and is scheduled at time $t + 1$ or later, where $t = 1$.

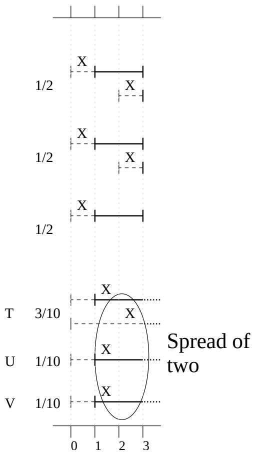


Figure 6: A three-processor PD^2 schedule using our method modified for MTTs with different-weight threads.

3.2 Method Applied to EDF

In considering EDF, we initially assume (as before) that each of an MTT's threads has the same weight, and that spread is as defined in the introduction. Under the conditions of Theorem 2 below, a thread can execute for at most $2 \cdot e_{max}$ consecutive time units, as shown in Fig. 7, which allows us to prove (in the appendix) the stated spread guarantee. Note that, in EDF, *jobs* become urgent, not subtasks.

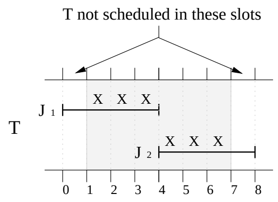


Figure 7: Assuming $\Delta = 0$, T executes for the maximum possible number of consecutive time units when consecutive jobs are scheduled as shown here.

Theorem 2 *Consider a task set τ for which tardiness is at most Δ under EDF, and let e_{max} denote the largest job execution cost in τ. If EDF is modified as described above for PD^2, but instead jobs (rather than subtasks) are allowed to become early-release eligible up to $2 \cdot e_{max}$ time units before* *their actual release times, and $T.p \geq T.e + 1 + \Delta$ for each $T \in \tau$, then the spread of any MTT is at most $2 \cdot e_{max} + 1$.*

As an aside, recall that tasks scheduled by EDF may miss their deadlines by bounded amounts, as stated in Sec. 2.2. Any tardiness arising from our method (by choosing to not shift the schedule as required) would be added to the tardiness bound for EDF.

Non-preemptive EDF. In non-preemptive EDF, lower-priority jobs can block higher-priority jobs for up to $e_{max} - 1$ time units. We already account for at least this level of disruption in most cases for preemptive EDF; however, additional disruption can occur because a thread can block other pending threads in its MTT. To account for this extra disruption, our spread guarantee must be increased by $e_{max} - 1$ time units, *i.e.*, to $3 \cdot e_{max}$.

Removing the "same-weight" restriction. In both variants of EDF, during each period, all jobs in the same MTT have the same release and deadline regardless of execution cost. Thus, referring back to our earlier discussion of PD^2, such jobs are "equivalent." We can therefore re-define spread for EDF in the case where the "same-weight" restriction is removed, as we did for PD^2, and with a few minor changes to the rules stated earlier, our results for both preemptive and non-preemptive EDF still hold.

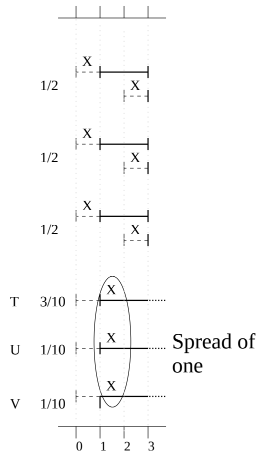


Figure 8: Task set from Fig. 6, scheduled with EDF.

The new definition of spread k is as follows: if a job T_j of some thread T of the MTT is scheduled at time t, then for each thread U of the MTT, where job U_j has not executed to completion, at least one time unit of execution of U_j is scheduled in the interval $[t, t + k)$. We can claim that at least one time unit of execution of U_j is scheduled due to our earlier requirement that the size of our time unit is set such that all execution costs and periods are integral. Note that this definition of spread provides nearly the same opportunities for cache reuse as before.

As an example, consider Fig. 8, which depicts the same task set considered earlier in Fig. 6. We allow early releasing of all jobs by one time unit. At time 1, jobs T_1, U_1, and V_1 are scheduled and execute for one time unit. This results in a spread of one for this MTT, since all jobs were scheduled concurrently. When job T_1 next executes it will achieve a spread of "one" by default, since all jobs of other task threads in the MTT either will have not been released or have run to completion. Note that while this example is for preemptive EDF, the schedule would only change slightly for non-preemptive EDF and would result in the same spread.

				Spread								
	Wt.			MTT Size = 2			MTT Size = 3			MTT Size = 4		
Algorithm	**Constr.**	X	**ER**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Max**
Reg. Pfair	$(0, 1/3]$	–	0	1	1.35	**41**	1	1.66	**40**	1	1.99	**41**
Mod. Pfair	$(0, 1/3]$	3	2	1	1.27	**2**	1	1.52	**2**	1	1.77	**3**
Reg. Pfair	$(0, 1/2]$	–	0	1	1.40	**37**	1	1.78	**41**	1	2.18	**37**
Mod. Pfair	$(0, 1/2]$	4	3	1	1.28	**2**	1	1.53	**2**	1	1.77	**3**
Reg. Pfair	$(0, 3/4]$	–	0	1	1.39	**25**	1	1.83	**33**	1	2.29	**41**
Mod. Pfair	$(0, 3/4]$	7	6	1	1.29	**2**	1	1.57	**2**	1	1.81	**3**

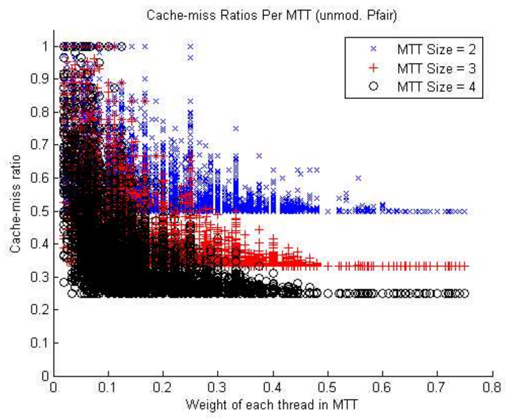
Table 1: Spread under PD^2. Each entry represents 50,000 task sets.

4 Experimental Results

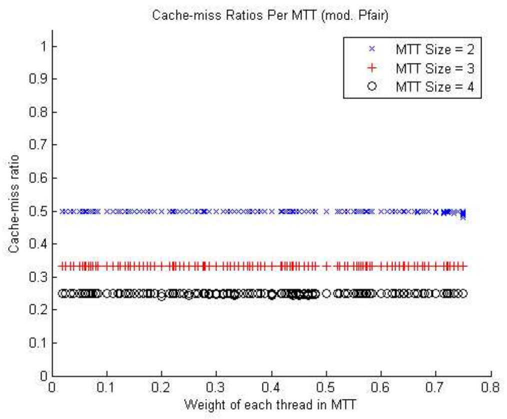
In this section, we assess the efficacy of our method in reducing spread and improving cache performance. With the exception of the last set of experiments below, the MTTs in our experiments have same-weight threads. Due to space constraints, we only consider PD^2 (though similar data could also be presented for EDF).

Spread reduction experiments. First, we randomly generated 50,000 task sets in several categories, and simulated the scheduling of these task sets on a four-core system. For each generated task set, we first allowed no early releasing and did not shift the schedule, and then allowed early releasing and shifted by $X - 1$ quanta, as in (2). An upper bound of $1/3$, $1/2$, or $3/4$ was enforced on task weights, depending on the experiment. Task periods varied from two (or three, if task weights could not exceed $1/3$) to 50. All task sets fully utilized all four cores, and MTTs varied in size from two to four (the total number of cores). These constraints permitted the inclusion of task sets with a wide variety of task weights, including those with large periods (*e.g.*, 50). The only types of tasks not included were tasks with weight exceeding $3/4$. All simulations were run up to the task-set hyperperiod. Results are shown in Table 1. As seen, our method always generates low average spreads (near one or two quanta), even lower than might be expected from the spread guarantees proved in Sec. 3. Note also that our method *always prevents extremely high spreads*, as shown in the boldface columns of Table 1.

L2 Cache Performance. We next demonstrate the effectiveness of our method in reducing L2 miss rates. We first estimated miss ratios for the same 50,000 task sets considered earlier using a simple (hand-coded) cache model. This model assumed a "best-case" scenario where the cache was fully associative. Each thread sequentially accessed 10,000 cache blocks, or 640K of memory, every quantum. The region of memory accessed was dependent on the subtask—equivalent subtasks of threads in the same MTT accessed the



(a)



(b)

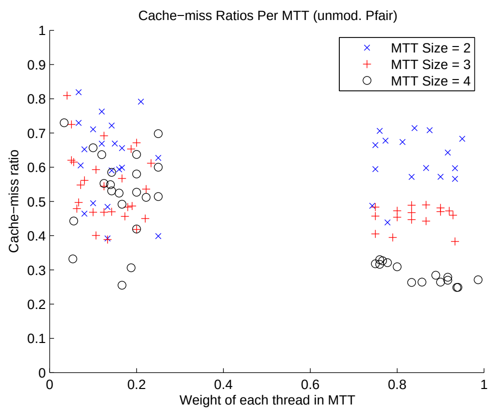
Figure 9: Cache-miss ratios for PD^2 both **(a)** without and **(b)** with our method, using a simulated simple cache model. Each scatter plot represents the same 50,000 task sets from Table 1 with weight constraint $(0, 3/4]$.

same unique region of memory. Thus, the only opportunities for cache reuse existed when threads of the same MTT accessed the same region of memory. We assumed that the amount of data accessed per subtask is the same every quantum regardless of cache performance: if a task finishes early, the rest of the quantum is wasted. We further assumed that the L2 cache can hold exactly four working sets of data, and used an LRU replacement policy. Cache blocks that could be reused during the current quantum were reused before being replaced (an idealistic assumption). Cache blocks that were not reused were eventually replaced per the LRU policy. This admittedly simplistic cache model allowed all tasks to be scheduled up to the task-set hyperperiod, as done in the prior experiment. This was not possible with the experiments discussed below, where a very exact (but slow) computer-architecture simulator was used. The results of experiments conducted assuming our simple cache model, shown in Fig. 9, are quite dramatic. Note that the best achievable cache-miss ratio for an MTT with N threads is $1/N$, and most MTTs approach this miss ratio with our method.

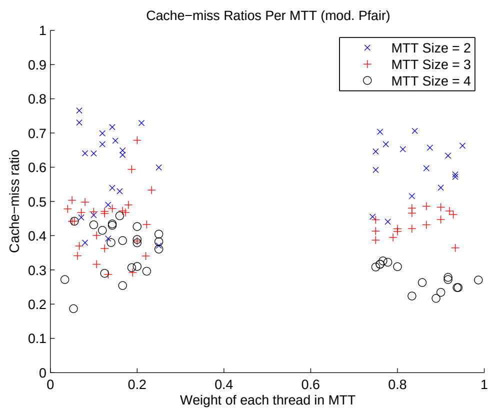
We next ran more realistic and complex experiments by using the SESC Simulator [13], which is capable of simulating a variety of multicore architectures. (Note that SESC executes *actual* task and scheduling code, and therefore scheduling, preemption, and migration costs were accounted for in these simulations.) In order to examine the benefits of our method per MTT, the simulator was modified so that each memory access could be "tagged" with a value indicating the MTT with which it was associated. The simulated architecture consisted of four cores, each with dedicated 16K L1 data (4-way set associative) and instruction (2-way set associative) caches with random and LRU replacement policies, respectively, and a shared 2048K 8-way set associative on-chip L2 cache with an LRU replacement policy. Each cache has a 64-byte line size. The memory access pattern of all threads remained the same as in the simpler experiments, and thus the L2 cache could hold approximately three working sets of data. Additionally, all threads in the same MTT start accessing memory regions from a different location, wrapping if necessary. This better utilizes the cache and prevents threads from proceeding in "lock step" while waiting for blocks to be loaded into the cache from main memory, resulting in virtually no cache benefit.

One application with such a memory-access pattern is parallel motion compensation search, which is the most compute-intensive part of MPEG-2 video encoding. Here, tasks access the same region of memory during the search but starting at different locations. Such an application might encode a video stream in real time on a frame-by-frame basis, and therefore would require (soft) real-time guarantees. Additionally, there would clearly be some benefit to co-scheduling tasks that are encoding the same frame (during the same quantum of computation).

In these experiments, we simulated 50 randomly-generated task sets for 20 quanta (instead of up to the hyperperiod) assuming a 0.75-ms quantum. While the SESC Simulator is very accurate, it comes at the cost of being quite slow. Therefore, longer and more detailed results could not be obtained because of the length of time it took the simulations to run. In order to demonstrate the substantial impact of our method on MTTs with low-weight threads, we required all tasks to either have weight at most $1/4$ or at least $3/4$. (This creates opportunities for heavier tasks to disrupt lighter tasks in the same MTT.) Task periods varied from three to 100, and all task sets fully utilized all four cores as before, with MTTs varying in size from two to four. In all cases, we only early release by six quanta—we would need to early release by much more in order to make spread *guarantees*, but we can still see substantial benefits with limited early releasing. Results are shown in Fig. 10. These realistic results,



(a)



(b)

Figure 10: Cache-miss ratios for PD^2 both **(a)** without and **(b)** with our method, using the SESC Simulator. Each scatter plot represents 50 task sets.

while not as dramatic, still demonstrate a significant benefit for lower-weight threads. Note the especially large benefit for MTTs of size three and four, where cache-miss ratios in the range of 50% to 75% decrease to at most 50% with few exceptions. Note also that opportunities for cache reuse are limited by our memory access pattern, and therefore all miss ratios are quite high. However, our method shows a substantial overall improvement with these task sets. Additionally, because an L2 miss incurs a time penalty up to two orders of magnitude greater than a hit, *a relatively small miss-rate difference can impact performance significantly*, as seen in [1].

Removing the "same-weight" restriction. To evaluate our method when scheduling MTTs with threads of varying weights, we ran on SESC a task set with seven $1/2$-weight tasks and one MTT containing four $4/41$-weight threads (a prime period was desired for these experiments), allowing early releasing by one quantum. The setup was exactly the same as in the prior SESC experiments, except that a 4096K L2 cache was used, task sets were run for 50 quanta instead of 20 to accommodate the large periods of the MTT threads, and the region of memory accessed was unique to same-deadline jobs in each MTT, rather than equivalent subtasks. Our method resulted in a substantial decrease of the L2 miss rate for the MTT, from 85.59% to 42.82%. We then modified the weights of the tasks in the MTT so that they gradually diverged in different ways. As we did this, the benefit of our method decreased; however, it was still substantial. With thread execution costs of $\{1, 3, 5, 7\}$ (evenly distributed), the miss-rate was reduced from 86.42% to 74.01%; with $\{1, 1, 1, 13\}$ (equal small, one large), from 89.20% to 71.10%; with $\{1, 5, 5, 5\}$ (equal large, one small), from 84.26% to 52.50%; with $\{1, 2, 3, 10\}$ (multiple small, one large), from 88.70% to 76.67%; and with $\{1, 4, 5, 6\}$ (multiple large, one small), from 85.74% to 72.44%. Note that we see less benefit for MTTs with fewer equal-weight threads.

5 Concluding Remarks

We have proposed a "spread-cognizant" scheduling method that decreases average and maximum spreads in both the PD^2 and global EDF scheduling algorithms. This method can be generalized to apply to any deadline-based global scheduling algorithm. We also presented an evaluation of our method that demonstrates its effectiveness in reducing spreads and lowering L2 miss rates.

There are several directions for future work. First, we want to combine this scheduling method with the methods in [1] so that both the "encouragement" and "discouragement" of co-scheduling can be supported in the same system. Second, we wish to include support for critical sections and precedence constraints in our work. Third, we want to investigate further any potential for a trade-off between the early-release interval length and the spread we can guarantee. Fourth, we would like to showcase our method by using it within applications on a real multicore system. Finally, the execution cost of a task is strongly dependent on its cache performance, which may depend on the spread guarantees made for its MTT. Those guarantees are, in turn, a function of task weights, and thus execution costs. Thus, there appears to be a "chicken and egg" problem that we need to investigate further. Timing analysis on multicore platforms is still in its infancy, and is a non-trivial area of future work upon which our work depends; however, if a timing analysis tool for multicore architectures were developed that could determine the execution cost of a task given the spread guarantees of the task set, then we may be able to avoid the "chicken and egg" problem by performing timing analysis for all tasks in the task set assuming a certain spread guarantee. If the execution costs calculated would guarantee such a spread, then the timing analysis is valid and we can assume the execution costs calculated are correct; otherwise, we can repeat the attempt with another spread guarantee. Naturally, as lower spreads imply lower cache-miss rates, thus implying reduced execution costs, it would be beneficial to choose the execution costs associated with the lowest valid spread guarantee.

References

[1] J. Anderson, J. Calandrino, and U. Devi. Real-time scheduling on multicore platforms. *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symp.*, pp. 179–190, 2006.

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

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

[4] G. Blelloch and P. Gibbons. Effectively sharing a cache among threads. *Proc. of the 16th ACM Symp. on Parallelism in Algs. & Archs.*, 2004.

[5] U. Devi and J. Anderson. Tardiness bounds for global EDF scheduling on a multiprocessor. *Proc. of the 26th IEEE Real-Time Systems Symp.*, pp. 330–341, 2005.

[6] C. Farivar. Intel Developers Forum roundup: four cores now, 80 cores later. http://www.engadget.com/2006/09/26/intel-developers-forum-roundup-four-cores-now-80-cores-later/, 2006.

[7] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Throughput-oriented scheduling on chip multithreading systems. Technical Report TR-17-04, Division of Engineering and Applied Sciences, Harvard University, 2004.

[8] A. Fedorova, M. Seltzer, and M. Smith. Cache-fair scheduling for chip multiprocessors. Manuscript, Harvard University, 2006.

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

[10] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning on a chip multiprocessor architecture. *Proc. of the 13th Int'l Conf. on Parallel Arch. and Comp. Techs.*, pp. 111–122, 2004.

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

[12] L. Peng, J. Song, S. Ge, Y.-K. Chen, V. Lee, J.-K. Peir, and B. Liang. Case studies: Memory behavior of multithreaded multimedia and AI applications. *Proc. of 7th Workshop on Computer Architecture Evaluation using Commercial Workloads*, 2004.

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

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

[15] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. *JCSS*, 72(6):1094–1117, 2006.

[16] P. Valente and G. Lipari. An upper bound to the lateness of soft real-time tasks scheduled by EDF on multiprocessors. *Proc. of the 26th IEEE Real-Time Systems Symp.*, pp. 311–320, 2005.

Appendix: Proofs of Theorems 1 and 2

Theorem 1. Recall that Theorem 1 deals with MTTs with same-weight threads. We prove that for any i, if t is the earliest time at which some subtask T_i is scheduled, where T is in MTT R, then all task threads in R have their i^{th} subtask scheduled in $[t, t+X)$. We henceforth use the term "thread" when referring to the tasks of some MTT, and the term "task" to refer to any task (which may or may not be from some MTT). We begin by stating a lemma concerning the execution rate of same-weight tasks, which follows easily by inducting over the subtask indices.

Lemma 1 *If U and V are threads in the same MTT, and if subtask U_j is scheduled at or after subtask V_k, then $k - j \leq 1$.*

Set categorization. At time t in a schedule $\mathcal{S}$, tasks are placed in the sets below—see Fig. 11. Subsets G_2 and H_2 contain threads with a subtask that is eligible at time t and must be scheduled over the interval $[t+1, t+X)$ to avoid a spread violation. Hereafter, we call such eligible subtasks *pending subtasks* (and, implicitly, this term is used with respect to time t).

- Set G: includes each thread $T \in R$, where R is any MTT, such that, for some i:
  - **(i)** No subtask U_i is scheduled before time t, for $U \in R$;
  - **(ii)** U_i is scheduled at time t for some thread $U \in R$;
  - **(iii)** U_{i-1} is scheduled at time t for some thread $U \in R$;
  - **(iv)** T_i or T_{i-1} is scheduled at time t.

  Set G is partitioned into subsets G_1 and G_2 where $T \in G_1$ iff T_i is scheduled at t, and $T \in G_2$ iff T_{i-1} is scheduled at t (implying T_i is not scheduled at t). Note that, by (ii), (iii), and Lemma 1, no subtask of a thread in MTT R with an index other than i or $i-1$ can be scheduled at time t.

- Set H: includes each thread $T \in R$, where R is any MTT, such that, for some i:
  - **(i)** No subtask U_i is scheduled before time t, for $U \in R$;
  - **(ii)** U_i is scheduled at time t for some thread $U \in R$;
  - **(iii)** Some thread $U \in R$ is not scheduled at time t.

  Set H is partitioned into disjoint subsets H_1 and H_2 where $T \in H_1$ iff T_i is scheduled at t, and $T \in H_2$ otherwise.

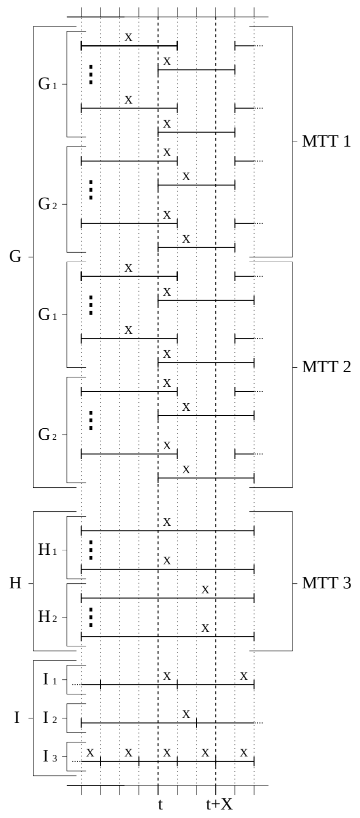


Figure 11: Sets G, H, and I, and their respective subsets.

Note that if U and U' are from the same MTT, and U_i is scheduled at t but U'_i could not be scheduled at t, then either U'_i's predecessor is scheduled at t, or U'_i is not among the M highest-priority subtasks selected for execution. Since we assume that all threads of an MTT have equal weight, by the *Priorities* rule, there is at most one MTT for which the latter could have happened. Thus, we have the following.

(PH) Set H contains threads from at most one MTT, and these threads have lower priority than the lowest-priority thread in set G at time t.

- Set I: includes each task T not in sets G or H. Set I contains disjoint subsets I_1, I_2, and I_3, defined as follows.
  - (**i**) $T \in I_1$ iff T is scheduled at time t, but not within $[t+1, t+X)$.
  - (**ii**) $T \in I_2$ iff T is not scheduled at t, but is within $[t+1, t+X)$.
  - (**iii**) $T \in I_3$ iff T is scheduled at t *and* within $[t+1, t+X)$.

If $G_2 \cup H_2$ is empty, then perfect parallelism is achieved for any subtasks from MTTs scheduled at time t. Thus, we assume:

(PE) $G_2 \cup H_2$ is non-empty.

In our proof, we assume there is a spread violation, which implies that t' defined next exists, and then derive a contradiction.

Definition 1: t' is the latest time at which any pending subtask of a thread in $G_2 \cup H_2$ is scheduled, where $t' \geq t+X$.

We assume the following for any task T in either I_2 or I_3.

(PI) A subtask of T has equal or higher priority than at least one pending subtask of a thread in $G_2 \cup H_2$ at some time u in the interval $[t+1, t'+1)$.

Otherwise, by the *Priorities* rule and Def. 1, T could not be scheduled until all pending subtasks of threads in $G_2 \cup H_2$ were scheduled, and we would not need to account for T in our proof.

Task allocations over an interval. For any subset $\alpha \in \{G_1, G_2, H_1, H_2, I_2, I_3\}$, we define $A_X(\alpha)$ to be the maximum number of subtasks T_k of any *one* task $T \in \alpha$, where $k > 0$, scheduled over the interval $[t+1, t+X)$. $A_{t'}(\alpha)$ is similarly defined with respect to $[t+1, t'+1)$, except that subtasks scheduled at time t' with priority lower than any pending subtask of a thread in $G_2 \cup H_2$ also scheduled at that time are not counted, as such subtasks do not interfere with the scheduling of any pending subtask. By Def. 1, we have the following.

(PT) For any α, $A_{t'}(\alpha) \geq A_X(\alpha)$.

Free processor allocations. The number of "free" processor allocations, F, available for pending subtasks of tasks in $G_2 \cup H_2$ over the interval $[t+1, t+X)$ is given by subtracting from $(X-1) \cdot M$ the maximum number of allocations to tasks in other groups, and the *additional* allocations (over the first) that may be made to tasks in $G_2 \cup H_2$ before all pending subtasks in $G_2 \cup H_2$ are scheduled. That is, $F \geq (X-1) \cdot M - A_X(I_2) \cdot |I_2| - A_X(I_3) \cdot |I_3| - A_X(G_1) \cdot |G_1| - (A_X(G_2)-1) \cdot |G_2| - A_X(H_1) \cdot |H_1| - (A_X(H_2)-1) \cdot |H_2|$. Our proof obligation is to show that $F \geq |G_2| + |H_2|$. By (PT),

$$\begin{aligned} F \;\geq\; & (X-1) \cdot M - A_X(I_2) \cdot |I_2| - A_X(I_3) \cdot |I_3| \\ & - A_X(G_1) \cdot |G_1| - (A_X(G_2)-1) \cdot |G_2| \\ & - A_{t'}(H_1) \cdot |H_1| - (A_{t'}(H_2)-1) \cdot |H_2|. \end{aligned} \quad (3)$$

The next lemma follows almost directly from the *Early-Release Eligibility* rule, and thus is stated without proof.

Lemma 2 *Suppose subtask T_i is released in the interval $[t+1, t+X)$ (i.e., $t+1 \leq r(T_i) < t+X$). If no prior subtask of task T is scheduled in $[t, r(T_i))$, and T_i is one of the e highest-priority subtasks at time t, where $e = M - (|\mathcal{U}| + |\mathcal{H}|)$ as defined by the* Early-Release Eligibility *rule, then T_i is early-release eligible in slot t. Additionally, if T_i is non-urgent in $[t+1, t')$, then T_i is not early-release eligible in $[t+1, t')$.*

The following lemma is used later to calculate F.

Lemma 3 *The properties below hold for M, and the variables and subsets defined above.*

(**a**) $|G_1| + |G_2| + |H_1| + |I_1| + |I_3| \leq M$.

(**b**) $|H_2| \leq M$.

(**c**) $A_X(I_2) = 0$.

(**d**) $A_{t'}(H_1) = 0$ *and* $A_{t'}(H_2) = 1$.

(**e**) $(A_X(I_3) = A_X(G_1) = A_X(G_2) \leq X-2) \Rightarrow (F \geq |H_2| + |G_2|)$.

Proof: Part (a) holds since all tasks in sets G_1, G_2, H_1, I_1, and I_3 are scheduled at time t by definition. Thus, these subsets together contain at most M tasks. By (PH) and (PM), $|H| \leq M$. Hence, $|H_2| \leq M$, and therefore part (b) holds.

We prove part (c) by proving that no task in I_2 can receive an allocation in $[t+1, t+X)$. Assume to the contrary that some subtask of a task in I_2 is scheduled in $[t+1, t+X)$. Let T_i denote the earliest-scheduled such subtask and assume that it is scheduled in slot u. Note that, because $T \in I_2$, T_i's predecessor is not scheduled in $[t, u)$. Now, if $r(T_i) < t+X$, then by (PI) and Lemma 2, T_i would be scheduled at time t, contradicting $T \in I_2$. Hence, $r(T_i) \geq t+X$. This implies that T_i was early-release eligible when it was scheduled. But by condition (iii) of the *Early-Release Eligibility* rule, this implies that all of the urgent subtasks in $G_2 \cup H_2$ are scheduled in $[t+1, u]$. However, this contradicts Def. 1.

We begin our proof of part (d) with the following claim, which follows easily from (PH).

Claim 1 *A task in set H_1 can receive no subtask allocations in the interval $[t+1, t')$. Also, a task in set H_2 can receive only one subtask allocation in the interval $[t+1, t')$—the allocation for its pending subtask.*

If H is empty, then (d) holds easily. Otherwise, by part (iii) of the definition of set H, H_2 is nonempty. Thus, by Claim 1, $A_{t'}(H_1) = 0$ and $A_{t'}(H_2) = 1$.

Finally, part (e) can be established as follows.

$$\begin{aligned}
F &\geq (X-1) \cdot M - A_X(I_3) \cdot |I_3| - A_X(G_1) \cdot |G_1| \\
&\quad -(A_X(G_2) - 1) \cdot |G_2| \quad \{\text{by (3), (c), and (d)}\} \\
&\geq M + |G_2| + (X-2) \cdot |H_1| + (X-2) \cdot |I_1| \\
&\quad \{\text{by (a) and } A_X(I_3) = A_X(G_1) = A_X(G_2) \leq X-2\} \\
&\geq M + |G_2| \\
&\geq |H_2| + |G_2| \quad \{\text{by (b)}\}
\end{aligned}$$

■

The next lemma concerns urgent subtasks.

Lemma 4 *Suppose a subtask T_i of a task T is urgent from time $t+1$ until it is scheduled, is scheduled in the interval $[t+1, t')$, and T_i is not the first subtask of a task T scheduled in the interval $[t+1, t')$. Then, T_i must be scheduled at time $r(T_i)$ or later,* i.e., *it cannot be scheduled before its release time.*

Proof: If T_i is scheduled at time t_u, where $t+1 \leq t_u < t'$, then by the *Urgent Tasks* rule, there must exist some subtask U_i in the same MTT that was both non-urgent and scheduled at time $t_{nu} < t_u$. If $t_{nu} > t$, then by Lemma 2, U_i is not early-release eligible in the interval $[t+1, t')$, and thus U_i and T_i must both be scheduled at time $r(U_i)$ or later. Since $r(U_i) = r(T_i)$, T_i must therefore be scheduled at time $r(T_i)$ or later. If $t_{nu} \leq t$, then T_i must be the first subtask of task T scheduled in the interval $[t+1, t')$, for otherwise, subtasks T_i and T_{i-1} would be scheduled at time $t+1$ or later, while subtasks U_i and U_{i-1} are scheduled at time t or earlier, since U_{i-1} must be scheduled earlier than U_i. This cannot be the case since it implies that at some time $t_{nu} \leq t$, U_i was scheduled instead of T_{i-1}, and by Lemma 1, T_{i-2} must have been scheduled before time t_{nu}, and therefore T_{i-1} must have been eligible at time t_{nu}. ■

The following lemma concerns the first urgent subtask of a task scheduled in the interval $[t, t+X)$.

Lemma 5 *If subtask T_i of a task T is urgent from time $t+1$ until it is scheduled, is the first subtask of T scheduled in the interval $[t+1, t+X)$, and T_i is scheduled early, then the maximum number of allocations that task T can receive over the interval $[t+1, t+X)$ is no more than what it could have received if T_i had not been scheduled early.*

Proof: In the absence of early releasing, a maximal allocation for T over $[t+1, t+X)$ occurs when every subtask of T released in this interval is scheduled in the first slot of its window. As seen in Fig. 12, early releasing T_i cannot increase this allocation. This is due to the fact that, by Lemma 2 and Lemma 4, T_i is the only subtask of T that can be scheduled early in the interval $[t+1, t')$, and therefore by Def. 1, in the interval $[t+1, t+X)$. ■

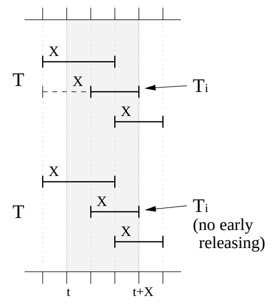


Figure 12: Lemma 5.

The following lemma is proved in [2].

Lemma 6 *Task T has windows of length $\left\lceil \frac{1}{wt(T)} \right\rceil$ or $\left\lceil \frac{1}{wt(T)} \right\rceil + 1$.*

The next lemma allows us to contradict Def. 1.

Lemma 7 *$F \geq |G_2| + |H_2|$, i.e., $t' < t + X$.*

Proof: Consider a subtask T_i of a task T such that $t+1 \leq r(T_i) < t + X$. If T_i is non-urgent over the entire interval $[t+1, t+X)$, then by Def. 1 and Lemma 2, early releasing is disabled for T_i over that same interval. The same is true if T_i is urgent at some time in the interval $[t+1, t+X)$, and therefore is urgent at some time in the interval $[t+1, t')$ by Def. 1, but is not the first subtask of T scheduled in the interval $[t+1, t')$, by Lemma 4. If T_i is urgent at some time in the interval $[t+1, t+X)$ *and* is the first task of T scheduled in the interval $[t+1, t+X)$ then T_i can be scheduled early, but by Lemma 5, T cannot receive any more allocations over the interval $[t+1, t+X)$ than if T_i had not been scheduled early. Together, these facts imply that we do not need to consider early-releasing over the interval $[t+1, t+X)$ when determining the maximum number of allocations a task can receive in that interval. We now consider three cases.

Case $W_{\max} \leq 1/3$. We establish $X = 3$ (a spread of three) in this case. By Lemma 6, $W_{\max} \leq 1/3$, and (1), no task can have a subtask window of length less than three, or an overlapping subtask window of length less than four. Thus, tasks in I_3 and G can receive at most two consecutive allocations before becoming ineligible, and thus no more than one additional allocation in the interval $[t+1, t+X)$, since they have already received an allocation in slot t. This is illustrated in Fig. 13. Thus, $A_X(I_3) = A_X(G_1) = A_X(G_2) = 1 \leq X - 2$, and by Lemma 3(e), $F \geq |G_2| + |H_2|$.

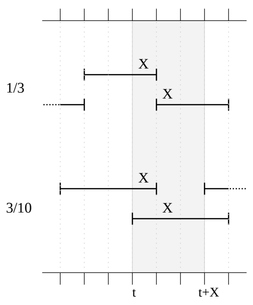


Figure 13: Lemma 7: $W_{\max} \leq 1/3$.

Case $1/3 < W_{\max} \leq 1/2$. We establish $X = 4$ (a spread of four) in this case. The reasoning is similar to the case above, and hence is omitted due to space constraints.

Case $W_{\max} > 1/2$. In this case, we consider a sequence $T_i, \ldots, T_j$ of subtasks of a task T of weight greater than 1/2 such that if any of $T_i, \ldots, T_j$ is scheduled in the last slot of its window, then each subsequent subtask in this sequence must be scheduled in its last slot (*e.g.*, T_1, T_2 or T_3, T_4, T_5 or T_6, T_7 in Fig. 14). In effect, $T_i, \ldots, T_j$ must be considered as a single schedulable entity subject to a *group deadline*, defined as $d(T_j) + 1$. Intuitively, if we imagine a job of T in which each subtask is scheduled in the first slot of its window, then the remaining empty slots correspond to the group deadlines of T. In Fig. 14, T has group deadlines at slots 4, 8, 11, 15, 19, and 22. The following claim follows from results concerning group deadlines proved in [2].

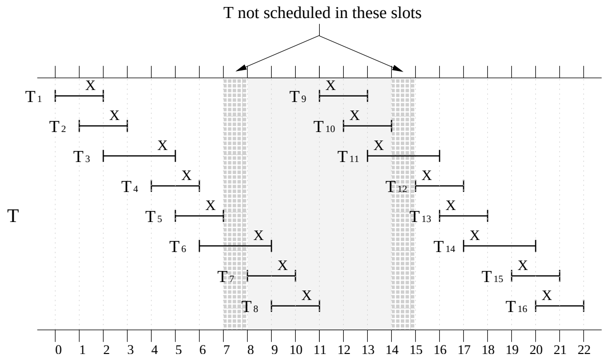


Figure 14: Maximum consecutive allocations to a task T of weight $8/11$.

Claim 2 *If $W_{\max} > 1/2$, then the maximum number of consecutive subtask allocations any task can receive is $2 \times \lceil \frac{1}{1-W_{\max}} \rceil - 2$.*

As an example, if $W_{\max} = wt(T)$ in Fig. 14, then a task can receive at most $2 \times \lceil \frac{1}{1-W_{\max}} \rceil - 2 = 2 \times \lceil \frac{1}{1-(8/11)} \rceil - 2 = 6$ consecutive allocations, demonstrated from time 8 to time 14.

By Claim 2, if X is $2 \times \lceil \frac{1}{1-W_{\max}} \rceil - 1$, then every task in I_3 and G is guaranteed to be ineligible for at least one quantum in the interval $[t+1, t+X)$ (again, in the absence of early releasing). Tasks in I_3 and G can receive no more than $2 \times \lceil \frac{1}{1-W_{\max}} \rceil - 3$ additional allocations in the interval $[t+1, t+X)$ since they have already received an allocation in slot t. Therefore, $A_X(I_3) = A_X(G_1) = A_X(G_2) = 2 \times \lceil \frac{1}{1-W_{\max}} \rceil - 3 \leq X - 2$. Thus, by Lemma 3(e), $F \geq |G_2| + |H_2|$. ■

From Lemma 7, Theorem 1 follows.

Theorem 2. This proof is similar to that for the $W_{\max} > 1/2$ case of PD^2, so only a sketch is provided. As with the PD^2 cases, we assume t' is defined as in Def. 1 and derive a contradiction by showing that $F \geq |G_2| + |H_2|$, given the X (spread) stated in Theorem 2. For EDF, the maximum number of consecutive time units that a task can execute is $2 \cdot e_{max}$. This is illustrated in Fig. 7, where $\Delta = 0$ is assumed. With a spread of $2 \cdot e_{max} + 1$, every task in I_3 and G is guaranteed to be ineligible for at least one time unit in the interval $[t+1, t+X)$ (again, in the absence of early releasing). This means that tasks in I_3 and G will execute for no more than $2 \cdot e_{max} - 1$ consecutive time units in the interval $[t+1, t+X)$ since they have already executed one time unit by time $t+1$. Therefore, $A_X(I_3) = A_X(G_1) = A_X(G_2) = 2 \cdot e_{max} - 1 \leq X - 2$. Thus, by Lemma 3(e), $F \geq |G_2| + |H_2|$.