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 certain groups of equal-utilization tasks to be scheduled together. This method can be applied to encourage tasks that share a common working set to be executed in parallel, which makes more effective use of on-chip shared caches. The proposed method can be adapted for use in *any* global, deadline-based scheduling approach. It exploits the fact that, under such approaches, "work" submitted at the same time by equal-utilization tasks will occupy consecutive slots in the scheduler's global priority queue, and thus, will be scheduled in close proximity, unless disrupted by later-arriving, higher-priority work. We show that certain properties of such global scheduling approaches can be exploited to limit this disruptive behavior.

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 [11].

In many proposed multicore platforms, different cores share on-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 widely-studied multicore architecture shown in Fig. 1, where all cores are symmetric, single-threaded, and share an L2 cache.

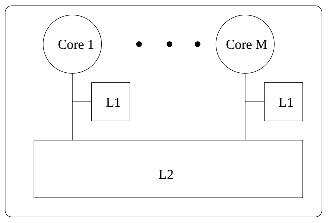


Figure 1: Multicore architecture.

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 sets of tasks can be *encouraged* to be co-scheduled—we refer to such a task set as a *task group*. Co-scheduling may be beneficial for tasks that share a common working set (WS). Fedorova *et al.* claimed that, for throughput-oriented applications, co-scheduling tasks that share a common WS would provide little performance benefit. The results of this paper, which are discussed below, call into question the validity of this claim for real-time systems. To simplify the problem, we henceforth assume that all tasks in each task group have the same *utilization*, or *weight*. This restriction also minimizes the duration of time that a parallel-accessed WS may create cache traffic. For example, with four tasks of weight $3/4$, $1/4$, $1/2$, and $1/2$, the WS is accessed at least 75% of the time, as opposed to 50% if the task weights were each $1/2$ and perfect parallelism ensured.

Related work. In work on (non-multicore) systems that support *simultaneous multithreading* (*SMT*), prior work on *symbiotic scheduling* is of relevance to our work [8, 9, 12]. In symbiotic scheduling, the goal is to maximize the overall "symbiosis factor," which 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. Additionally, the issue of scheduling tasks that require multiple processors has been considered previously [4]. However, this prior work places restrictions on scheduling, such as pinning certain tasks to certain processors, and provides few analytical results.

Proposed approach. The essence of the problem at hand is to encourage *parallelism*: when one of a group of tasks is scheduled, *all* such tasks should be scheduled. Unfortunately, achieving perfect parallelism for all task groups is not always possible. For example, it is not possible to schedule two parallel tasks of weight 1/2 together with a task of weight 3/4 on two processors without missing a deadline. 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 our efforts on minimizing a factor we call *spread*: if a task group has a spread of

[*]Work supported by NSF grants CCR 0309825 and CCR 0408996. A preliminary version of this paper was among a set of RTSS 2005 WIP papers selected to appear in ACM SIGBED Review.

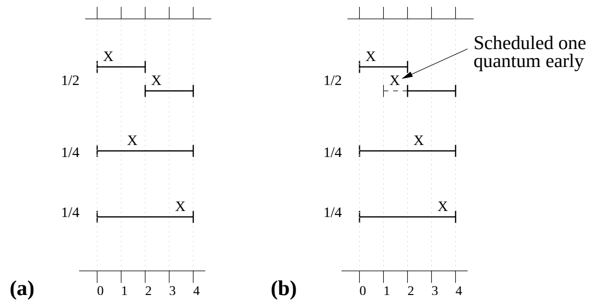


Figure 2: A one-processor Pfair schedule for a set of three tasks (of weight 1/2, 1/4, and 1/4, respectively) with **(a)** no early releasing and **(b)** early releases allowed by one quantum. This schedule can also be viewed as an EDF schedule where the execution cost of each job is one quantum. In depicting Pfair schedules, we use solid lines to indicate subtask windows, dashed lines to indicate where early releasing is allowed, and an "X" to indicate when a subtask is scheduled.

k, then the i^{th} quantum of computation of each task in the group must be scheduled within an interval $[t, t+k)$ for some t (treating each quantum as a "time unit"). A spread of one is perfect parallelism. Our goal, then, is to schedule tasks so that real-time constraints are met and spread is minimized to the extent possible. Note that, even with small spreads that exceed one, there may be a potential for cache reuse. Although our motivation here is better cache performance, minimizing spread may also be beneficial for synchronization: if groups of tasks have precedence constraints or are otherwise dependent on one another, then small spreads may be desirable. 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" submitted at the same time by same-weight tasks 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 completely *optional* as to whether the scheduler considers it to be available for execution. We can exploit this fact to minimize disruptions to task groups caused by higher-priority work.

As an example of the last point, consider the Pfair schedules in Fig. 3, where both tasks of weight $1/4$ are placed in their own task group. Inset (a) shows a schedule without early releasing. In inset (b), all subtask windows are shifted right by one quantum, and all tasks are early 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), $k = 1$. Both schedules result in a spread of three for the $1/4$-wt. task group. 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 insets (b) and (c), we can instead make it completely optional whether to schedule subtasks for the first k quanta after their release, and allow jobs to miss their deadlines 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 particular task set, then we could choose instead to create a 3-shifted schedule and allow jobs to early release by only three quanta, but also make it optional whether to schedule jobs during the first quantum after their actual release. In this case, deadlines could be missed by at most one quantum. In general, 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 encouraging low spreads that can be adapted to any deadline-based, global scheduling approach. We present these rules in Sec. 3.1 by focusing on PD^2 and then explain in Sec. 3.3 how they can be adapted to apply to EDF. Experiments involving randomly-generated task sets presented in Sec. 4 show that these rules are effective in reducing spreads in the *average case*. In fact, in these experiments, spreads were usually very close to one (which is perfect parallelism). However, in real-time systems, spread *guarantees* are desirable. We show in Sec. 3 that such guarantees are possible (for both PD^2 and EDF) and that they hinge on the largest task weight in the system. As a final contribution, we assess the effectiveness of our spread-minimization method by examining the results of experiments involving several real-time workloads conducted on a multicore simulator. In these experiments, the usage of our method resulted in L2 miss-rate reductions in the range of 8–35%. We argue that our method *can*, but does not always, lead to lower L2 miss rates. Determining this for a particular application can be difficult due to the inherent complexity of memory access patterns.

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 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 [5, 13] and EDF scheduling. For simplicity, we consider only periodic task sys-

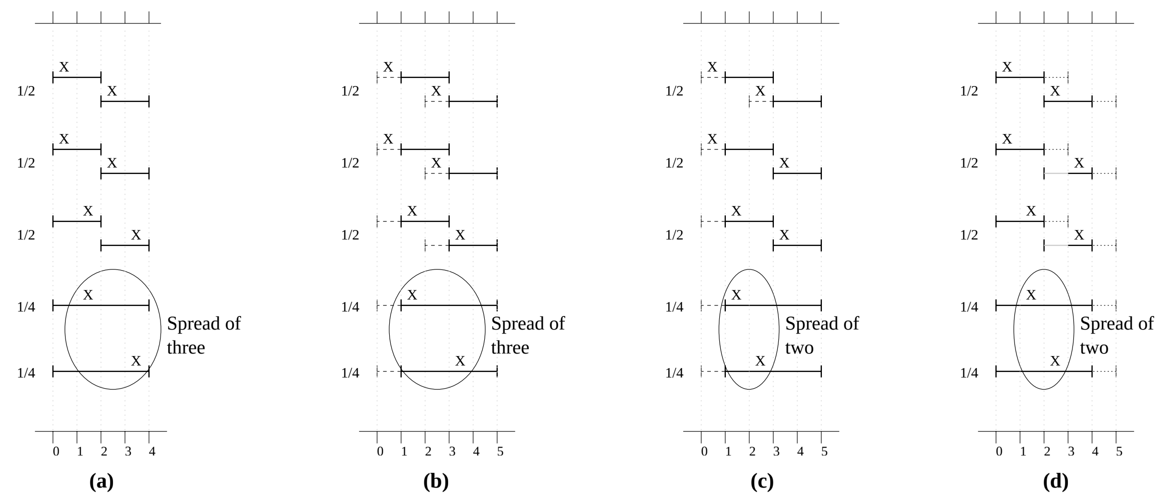


Figure 3: A two-processor Pfair schedule of a set of five tasks (three of weight 1/2, and two of weight 1/4) with **(a)** no early releasing; **(b)** early releasing by one quantum and all windows right-shifted by one quantum; **(c)** similarly-shifted windows, but selective early releasing; **(d)** 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.

tems. However, all of our results apply to sporadic task systems, and also to intra-sporadic task systems, as considered in work on Pfair scheduling [13]. In a periodic task system τ, each task T is subdivided into a sequence of sequential *jobs*, and is characterized by two quantities: 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. A time unit is referred to as a *quantum*. Each task's execution cost and period are each assumed to be integral values, where a value of one corresponds to the length of the system's quantum size. The quantity $T.e/T.p$ is the *weight*, or *utilization*, of T, denoted $wt(T)$. The problem of interest to us is that of scheduling a set of periodic tasks on $M \geq 1$ processors (or cores).

2.1 Pfair Scheduling

Pfair scheduling algorithms [5, 13] (as applied to periodic tasks) 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.

Processor time is allocated one quantum at a time by subdividing 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.

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

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. Fig. 4 shows an example. A task's windowing is defined so that its allocation approximates that of an ideal (fluid) system that allocates $wt(T) \cdot L$ units of processor time to each task T in any interval of length L. Note that, since all tasks in each task group have the same weight, tasks in the same group always have subtask windows and task periods that coincide. (For sporadic or intra-sporadic tasks, this requirement is needed as well.)

Pfair scheduling algorithms. Pfair scheduling algorithms schedule tasks by scheduling their 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 [3, 13], which uses two tie-breaking rules. Because it is optimal, PD^2 meets all subtask deadlines, as long as the system's total utilization, $\sum_{T \in \tau} wt(T)$, does not exceed M.

Tardiness bounds. If a scheduling algorithm cannot ensure all timing requirements, then it may still be possible to show that deadline tardiness is always bounded by some number of quanta. A *tardiness bound* of B means that a subtask will complete at most B quanta after its deadline.

Early-release scheduling. In the *early-release* (*ER*) *task model* [3], a subtask may be released early, *i.e.* be eligible to execute before its window, while still ensuring the timing requirements of all tasks. The decision of whether or not to release a subtask early, and how early to release it, can be arbitrary, except that all subtasks of a task must execute in order.

2.2 EDF Scheduling

In global earliest-deadline-first (EDF) scheduling, jobs are scheduled in the order of increasing deadlines, with ties broken arbitrarily. In the case of EDF scheduling, all task groups are assumed to consist of tasks with the same execution cost and period. As

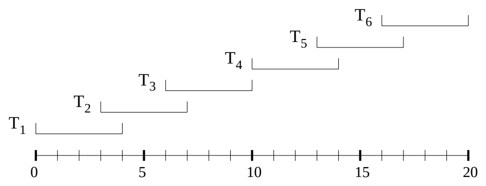


Figure 4: Windowing for a task with weight $wt(T) = 3/10$ under Pfair scheduling over the interval [0,20]. Note that three (six) subtasks have deadlines by time 10 (20). Thus, if each subtask meets its deadline, then T's allocation up to these times is the same as in an ideal system ($\frac{3}{10} \times 10$ and $\frac{3}{10} \times 20$, respectively).

such, all such periods will always coincide. (For sporadic tasks, this requirement is needed as well.)

EDF scheduling is *not* optimal, and therefore tasks may miss their deadlines even when total utilization is at most M. It has been shown, however, that deadline tardiness under EDF is bounded [6] when the total utilization of a task set is not restricted. Additionally, jobs may be optionally and arbitrarily early released under EDF, as in Pfair scheduling, with no additional tardiness penalties.

We assume a *quantized* version of EDF in this paper, where scheduling decisions are made only on quantum boundaries that are aligned across all processors (and all task execution costs and periods are integral, as noted earlier). We further assume that jobs of all tasks require their full execution cost. For example, a task with execution cost 4 will never have a job that completes after only three quanta. These assumptions are motivated by the fact that spread is a per-quantum measure. Additionally, if jobs can receive *fewer* allocations than their execution cost specifies, then we are essentially trying to provide spread guarantees for a group of tasks with varying execution requirements. This is a hard problem left as an area of future work.

3 Spread-Cognizant Scheduling

We begin with a description of our scheduling method applied to PD^2, and follow with a correctness proof of the spread guarantees we claim when applying this method. We then discuss how to adapt the method to *quantized* EDF.

3.1 Method Applied to PD^2

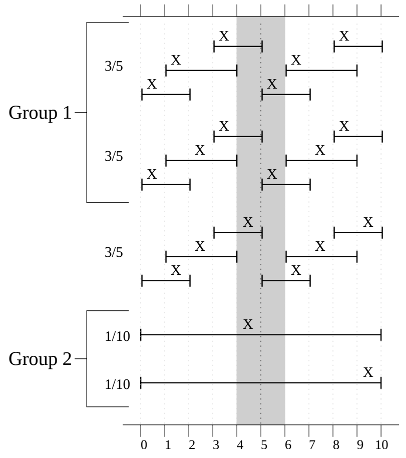
In general, we will use X to denote the spread guarantee we seek to establish. For PD^2, X is defined as follows, where $W_{\max} = \max_{T \in \tau} wt(T)$.

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

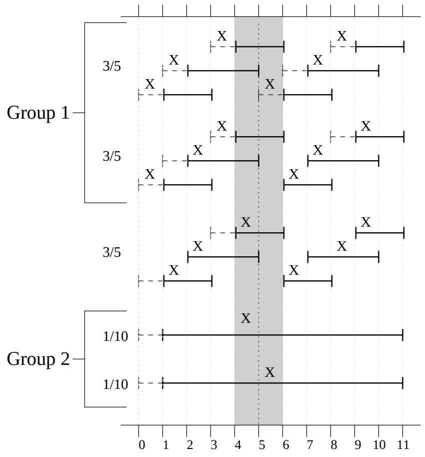
We describe our method as additional rules for PD^2. We assume we are working with an $(X-1)$-shifted schedule (or alternately, that subtasks can miss their deadlines by up to $X-1$ quanta—see Fig. 3(d)). Three rules are required:

- **Urgent Tasks.** When subtask T_i is scheduled, where T is in some task group R, and T is the first task in R whose i^{th} subtask is scheduled, each subtask U_i, where $U \in R$ and $U \neq T$, is flagged "urgent" until it is also scheduled.
- **Early-Release Eligibility.** A subtask T_j non-urgent at time t is "early-release eligible" at t if all of the following are true:
  - (**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 highest-priority subtasks at time t satisfying conditions (i) and (ii) above, where $e = M - (|\mathcal{U}| + |\mathcal{H}|)$.

  A subtask T_j that is urgent at time t is "early-release eligible" at time t if only 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 task-group identity used as a tie-break. (This ensures that task groups achieve the lowest possible spread when nothing in the PD^2 priority rules would prevent it.)



(a)



(b)

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

Fig. 5 shows how selective early-release eligibility reduces spread. In this example, early-release eligibility starts only *one* slot before the actual release time of a subtask—in many cases, even very limited early releasing has a substantial impact on spread. With regular PD^2 (inset (a)), the task set achieves maximum spreads of two and six for Groups 1 and 2, respectively. Our rules reduce the spread of Group 2 to two (inset (b)), without changing the spread of Group 1. This reduction happens because at time 5 in (b), the $1/10$-wt. task not scheduled at that time in (a) is favored over the $3/5$-wt. tasks by the *Urgent Tasks* rule. Note that the *Early-Release Eligibility* rule only allows one of the $3/5$-wt. 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 task was scheduled first, and Group 2 would still have a spread of two.

3.2 Correctness Proof

We now prove that when our method is applied to PD^2, spreads as defined by (2) are ensured. Specifically, we prove that, for any i, if t is the earliest time at which some subtask T_i is scheduled, where T is in task group R, then all tasks in R have their i^{th} subtask scheduled in $[t, t+X)$. We begin with the following property about all task groups.

(PM) A task group contains at most M tasks, which is the maximum parallelism achievable on an M-core architecture.

The next lemma precludes subtasks of two tasks from the same group being scheduled at the same time if their indices differ by more than one.

Lemma 1 *If tasks U and V are both in task group R, and if subtask U_j is scheduled at time t_U and subtask V_k is scheduled at time $t \le t_U$, then $k - j \le 1$.*

Proof: Follows easily by inducting over the subtask indices since tasks in the same group have the same window structure. ∎

Set categorization. At time t in a schedule $\mathcal{S}$, tasks are placed in the following sets, which are illustrated in Fig. 6. Subsets G_2 and H_2 contain tasks 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 task $T \in R$, where R is any task group, 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 task $U \in R$;
  - (**iii**) U_{i-1} is scheduled at time t for some task $U \in R$;
  - (**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 task in group R with an index other than i or $i-1$ can be scheduled at time t.
- Set H: includes each task $T \in R$, where R is any task group, such that, for some i:
  - (**i**) No U_i is scheduled before time t, where $U \in R$;
  - (**ii**) U_i is scheduled at time t for some task $U \in R$;
  - (**iii**) Some task $U \in R$ 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.

  Note that if U and U' are from the same group, 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. By the *Priorities* rule, there is at most one task group for which the latter could have happened. Thus, we have the following.

  (PH) Set H contains tasks from at most one task group, and these tasks have lower priority than the lowest-priority task 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 in the interval $[t+1, t+X)$.
  - (**ii**) $T \in I_2$ iff T is not scheduled at time t, but is scheduled in the interval $[t+1, t+X)$.
  - (**iii**) $T \in I_3$ iff T is scheduled at time t *and* in the interval $[t+1, t+X)$.

By the definitions of sets G and H, if $G_2 \cup H_2$ is empty, then H is empty and any task group satisfying conditions (i) and (ii) of set G achieves perfect parallelism for subtask i, and thus satisfies any spread guarantees we make. Therefore, we assume the following.

(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 task in $G_2 \cup H_2$ is scheduled, where $t' \ge t + X$.

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 task 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 tasks in $G_2 \cup H_2$ were scheduled, and we would not need to account for allocations to T in our proof, or T would be in I_1.

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 task 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, $A_{t'}(\alpha) \ge A_X(\alpha)$.

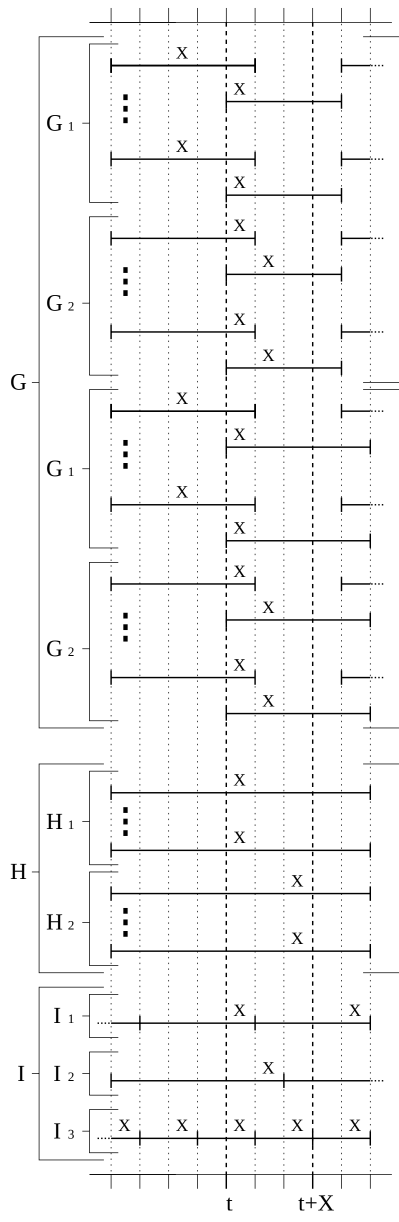


Figure 6: Sets G, H, and I, and their respective subsets.

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. That is, $F = (X-1) \cdot M - A_X(I_2) \cdot |I_2| - A_X(I_3) \cdot |I_3| - A_X(G_1) \cdot |G_1| - (A_X(G_2)-1) \cdot |G_2| - A_X(H_1) \cdot |H_1| - (A_X(H_2)-1) \cdot |H_2|$. 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 concerns early-release eligibility at or after t.

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 $[t+1, t')$.*

Proof: Assume that T_i is released in the interval $[t+1, t+X)$, and no prior subtask of task T is scheduled in the interval $[t, r(T_i))$. Then, T_i satisfies conditions (i) and (ii) of the *Early-Release Eligibility* rule. If we further assume that T_i is one of the e highest-priority tasks at time t, where e is defined in the statement of the lemma, then condition (iv) of the rule is satisfied, and it must also follow that $e > 0$. This implies that $M - (|\mathcal{U}| + |\mathcal{H}|) > 0$, and therefore $(|\mathcal{U}| + |\mathcal{H}|) < M$, so condition (iii) is satisfied. Thus, T_i is early-release eligible.

Once the eligible subtask of every task in $G_1 \cup H_1$ is scheduled in slot t, by the definition of subsets G_2 and H_2, each pending subtask of a task in $G_2 \cup H_2$ is both eligible and urgent from time $t+1$ until it is scheduled. By the *Early-Release Eligibility* rule, no subtask released in the interval $[t+1, t+X)$, and non-urgent over that same interval, can be early-release eligible at time $u > t$ unless $|\mathcal{U}| + |\mathcal{H}| < M$ for slot u, which cannot be the case for any slot in $[t+1, t')$. ■

The following lemma states a number of properties related to the calculation of 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), $|H| \leq M$ and $|H_2| \leq M$, 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.

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

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 task group 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 $t_{nu} < t$, 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 $[t, t+X)$.

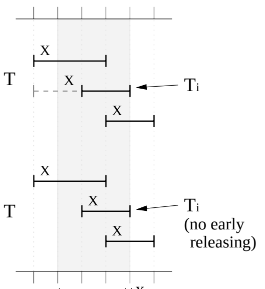


Figure 7: Lemma 5.

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. 7, 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 $[t+1, t+X)$. ■

The next lemma concerns subtask window lengths.

Lemma 6 (from [3]) *The length of any subtask window of a task T is either $\left\lceil \frac{1}{wt(T)} \right\rceil$ or $\left\lceil \frac{1}{wt(T)} \right\rceil + 1$.*

We now state our main lemma for algorithm PD^2, which 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, depending on the value of $W_{\max}$.

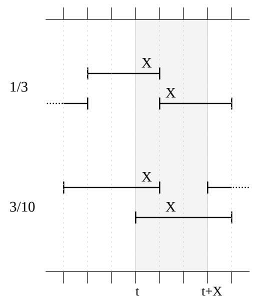


Figure 8: Lemma 7: $W_{\max} \leq 1/3$.

Case $W_{\max} \leq 1/3$. We claim a spread of three in this case, *i.e.*, $X = 3$. Lemma 6 and the upper bound on task weights, in conjunction with (1), imply that no task can have a subtask window of length less than three, or an overlapping subtask window of length less than four. Therefore, tasks in I_3 and G can receive at most two consecutive processor 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. 8. Therefore, $A_X(I_3) = A_X(G_1) = A_X(G_2) = 1 \leq X-2$. Thus, by Lemma 3(e), $F \geq |G_2| + |H_2|$.

Case $1/3 < W_{\max} \leq 1/2$. We claim a spread of four in this case, *i.e.*, $X = 4$. The reasoning is very similar to that 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 such that the windows of subtasks $T_{i+1}, \ldots, T_j$ are of length two and overlap the windows of their successor subtasks by one, and the window of subtask T_{j+1} is either of length three, or of length two and does not overlap with its successor subtask (*e.g.*, T_1, T_2 or T_3, T_4, T_5 or T_6, T_7 in Fig. 9). 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. 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

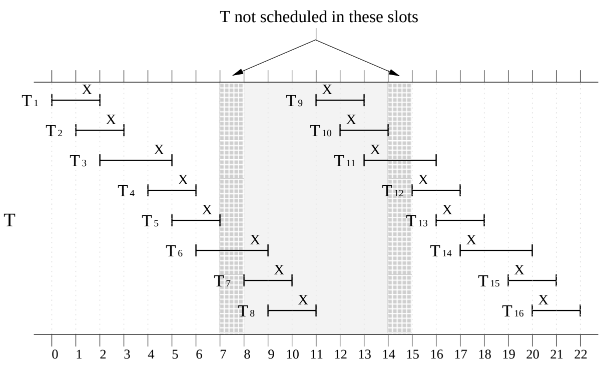


Figure 9: The maximum number of consecutive subtask allocations to task T is twice the maximum distance between consecutive group deadlines minus two. T has weight $8/11$.

subtask is scheduled in the first slot of its window, then the remaining empty slots correspond to the group deadlines of T. In Fig. 9, T has group deadlines at slots 4, 8, 11, 15, 19, and 22.

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

Proof: The calculation required here is straightforward and hence is omitted due to space constraints. (The term $\lceil \frac{1}{1-W_{\max}} \rceil$ is the maximum distance between consecutive group deadlines and is calculated in [3].) As an example, if $W_{\max} = wt(T)$ in Fig. 9, then the maximum number of consecutive subtask allocations for any task is $2 \times \lceil \frac{1}{1-W_{\max}} \rceil - 2 = 2 \times \lceil \frac{1}{1-(8/11)} \rceil - 2 = 6$, demonstrated from time 8 to time 14. ■

By Claim 2, if we increase our maximum guaranteed spread X to $2 \times \lceil \frac{1}{1-W_{\max}} \rceil - 1$, 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 1 *If PD^2 is modified as described above, and subtasks are early-release eligible $X-1$ quanta before their actual release times, then the spread of any group of tasks that we wish to co-schedule is no greater than X as defined in* (2).

3.3 Method Applied to EDF

Theorem 2 below states a spread result for our *quantized* version of EDF as described in Sec. 2.2, with aligned quanta and integral task execution costs and periods. The spread guarantees made for EDF are with respect to a *quantum of computation* rather than a *subtask* as in PD^2, and therefore we are concerned with the next *pending unit of computation* of each task in $G_2 \cup H_2$ rather than *pending subtask*. Additionally, in EDF, jobs become urgent instead of subtasks. Finally, note that in order for the method to be most effective, deadline ties should be broken in factor of the task with higher weight *before* urgency is used as a tie-breaker.

As an aside, note that tasks in task sets scheduled with EDF may miss their deadlines by bounded amounts, as stated in Sec. 2.2. Thus, tasks scheduled with EDF using our method, but without early releasing and a shifted schedule, may miss their deadlines by the sum of the tardiness bound for EDF *plus* any additional amount added by choosing to shift the schedule and early release by a smaller amount (or not at all).

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

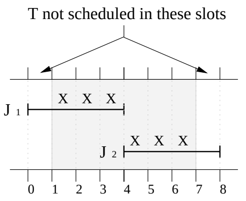


Figure 10: Assuming $\Delta = 0$, the maximum number of consecutive quantum allocations to task T occurs when consecutive jobs are scheduled as shown here.

Proof: 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 quantized EDF, the maximum number of quanta for which a task can consecutively receive processor allocations is $2 \cdot e_{max}$. This is illustrated in Fig. 10, 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 quantum in the interval $[t+1, t+X)$ (again, in the absence of early releasing). This means that tasks in I_3 and G can receive no more than $2 \cdot e_{max} - 1$ additional allocations in the interval $[t+1, t+X)$ since they have already received an allocation in slot t. Therefore, $A_X(I_3) = A_X(G_1) = A_X(G_2) = 2 \cdot e_{max} - 1 \leq X - 2$. Thus, by Lemma 3(e), $F \geq |G_2| + |H_2|$. ■

4 Experimental Results

In this section, we assess the efficacy of our method in reducing spread and achieving better L2 cache performance.

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-processor system with EDF and PD^2. For each task set with which we experimented, we first allowed no early-releasing and did not shift the schedule, and then allowed early-releasing and shifted by $X-1$ quanta, as specified earlier for both PD^2 and EDF. For PD^2, an upper bound on task weights was enforced, during task set generation, of $1/3$, $1/2$, or $3/4$, depending on the experiment. For EDF, all tasks had an execution cost of 1. Task periods varied from two (or three, if tasks could not have a weight of $1/2$) to 50. All task sets fully utilized all four processors, and the task groups varied in size from one (*i.e.*, a lone task in its own task "group") to four (the total

				Spread								
				Grp. Size = 2			Grp. Size = 3			Grp. Size = 4		
Algorithm	**Wt. Constr.**	X	**ER**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Max**
Reg. Pfair	$(0, 1/3]$	N/A	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	$(0, 1/2]$	N/A	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	$(0, 3/4]$	N/A	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**

(a)

				Spread								
				Grp. Size = 2			Grp. Size = 3			Grp. Size = 4		
Algorithm	e_{max}	X	**ER**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Max**
Reg. EDF	1	N/A	0	1	1.96	**41**	1	2.28	**42**	1	2.54	**45**
Mod. EDF	1	3	2	1	1.34	**2**	1	1.47	**2**	1	1.62	**2**

(b)

Table 1: Raw parallelism and spread for **(a)** Pfair and **(b)** EDF algorithms. Each different constraint represents 50,000 task sets.

number of processors). These constraints were reasonable as they included 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 greater than $3/4$ in PD^2 or execution cost over 1 in EDF. All simulations were run for the length of the task-set hyperperiod. Results are shown in Table 1.

Results. Our method generates low spreads in all cases (near two quanta), thus appearing to perform considerably better in practice than might be expected from the analytical upper bounds on spread proven in Sec. 3. For the task sets used in the global EDF experiments, $T.p \geq T.e + 1 + \Delta$ was *not* explicitly ensured for each task T in the EDF task sets. However, the spread results are still impressive and within the analytical bound stated in Sec. 3.3. Note that our method decreases average spread and *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 at improving L2 cache performance by simulating the scheduling and execution of task sets with the SESC Simulator [10], which is capable of simulating a variety of multicore architectures. We chose to use a simulator so that we could experiment with more cores than commonly available today. The simulated architecture consists of four cores, each with dedicated 16K L1 data (4-way set assoc.) and instruction (2-way set assoc.) caches with random and LRU replacement policies, respectively, and a shared 8-way set assoc. on-chip L2 cache with an LRU replacement policy (size varies per experiment). Each cache has a 64-byte line size.

All tasks in the same group access the same large "working set" region of memory. Tasks make one pass over a region of their working set per quantum of execution, and therefore only tasks in the same group can utilize the cache by reusing blocks already brought in by other tasks in that group. To encourage such reuse, all tasks in the same group access the same region of memory for a given quantum of computation, and each task starts accessing the region in a different location, wrapping if necessary. If all tasks started accessing memory in the same location, all tasks would proceed in a "lock step" manner while waiting for blocks to be loaded into the cache from main memory, resulting in virtually no benefit from the cache. By starting in different locations, tasks in the group can better reuse what remains in the cache later in the quantum.

The region of memory accessed each quantum is determined by a "sliding window" that moves by 15,000 cache blocks over the "working set" region of memory shared by all tasks in the group. This memory access pattern is intended to account for the worst-case memory performance of tasks and keeps groups of tasks synchronized with one another. Due to this memory access pattern, tasks access approximately 960K of memory per quantum. Thus, the sliding window allows for some level of cache reuse that decreases and eventually drops off entirely with increasing spread, due to capacity and conflict misses.

One application with the potential to exhibit a memory access pattern similar to the one described might be parallel motion compensation search, which is the most compute-intensive part of MPEG-2 video encoding. In such an application, some number of tasks would access the same region of memory during the search. However, each task would start accessing the region in different location. 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).

Hand-Crafted Task Sets. We created several hand-crafted task sets to demonstrate the effectiveness of our method at reducing L2 cache-miss rates. The hand-crafted task sets are listed in Table 2. We allowed early-releasing and shifted the schedule by the indicated number of quanta when applicable. Each task set was run for 20 quanta (assuming a 0.75-ms quantum length) on an architecture with the specified number of cores and the indicated L2 cache size. Table 3 shows for each case the L2 cache-miss rates that were observed.

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. Additionally, space constraints were also a limiting factor in the amount of experimental data we could present. We hope to present more extensive experiments in a future paper.

The L2 cache-miss rates given in Table 3 show that our method results in substantially better performance. Note that the opportunities for cache reuse are limited by our memory access pattern, and therefore all miss ratios are quite high. However, our method

Name	**No. Tasks**	**Task Properties**	**ER**	**No. Cores**	**L2 Size**
BASIC	5	2 of Wt. 1/4 (same group) 3 of Wt. 1/2 (indep.)	1	2	2048K
LONGER_BASIC	5	2 of Wt. 1/10 (same group) 3 of Wt. 3/5 (indep.)	1	2	2048K
ONE_PROC	3	2 of Wt. 1/4 (same group) 1 of Wt. 1/2 (indep.)	1	1	1024K
MAX_PARA	3	16 of Wt. 1/4 (4 groups of 4 tasks)	2	4	2048K
NO_PARA	5	16 of Wt. 1/4 (indep.)	2	4	2048K

Table 2: Properties of hand-crafted task sets.

Name	Reg. Pfair	Mod. Pfair	Reg. EDF	Mod. EDF
BASIC	79.57%	63.79%	79.18%	63.74%
LONGER_BASIC	60.54%	55.36%	40.27%	34.89%
ONE_PROC	80.93%	52.53%	80.92%	52.60%
MAX_PARA	23.43%	23.35%	23.45%	23.46%
NO_PARA	79.02%	78.73%	78.72%	79.02%

Table 3: L2 cache miss ratios for hand-crafted task sets.

shows a substantial overall improvement with these task sets. In task sets BASIC, LONGER_BASIC, and ONE_PROC, one task group can benefit from the cache, and tasks in that group have relatively low weight. In order to see a significant overall benefit, the actual performance of tasks in that group must have dramatically improved, though the SESC Simulator gave us no way of measuring the cache performance of specific tasks. Additionally, because an L2 miss incurs a time penalty roughly two orders of magnitude greater than a hit, *a miss-rate difference can correspond to a significant difference in performance*, as seen in [1].

BASIC and LONGER_BASIC consist of three independent tasks with large weight, and two tasks of smaller weight in the same task group. A spread of three is achieved for the task group of each set without our method, and thus little opportunity for cache reuse exists. With our method, spread is reduced to two, and the cache is better utilized, resulting in decreased miss rates. Note that LONGER_BASIC shows less of an overall improvement, as the tasks in the group are of lower weight and therefore there is a smaller improvement in overall cache performance.

ONE_PROC consists of one independent task with weight $1/2$, and two tasks of weight $1/4$ in the same task group. As there is only one processor, these two tasks cannot be co-scheduled, but we can achieve a spread of two with our method. This allows one task in the group to directly reuse data brought in from the other task during the preceding quantum, and we therefore see the most impressive benefit from reducing spread in this case.

MAX_PARA and NO_PARA demonstrate the dramatic cache benefits that can be achieved when tasks that share memory regions achieve a spread of one. Every quantum with set MAX_PARA, each block of memory brought into the cache is reused three times, resulting in approximately a 25% cache-miss ratio in all cases. Alternately, NO_PARA provides no opportunity for reuse, and we see very high cache-miss ratios as a result.

We emphasize that these example task sets demonstrate that our method *can* lead to lower L2 miss rates. However, there will not *always* be a benefit, and this is why it is difficult to demonstrate a benefit for some number of randomly-generated task sets, as was done in the spread reduction experiments presented earlier. Determining whether our method will improve cache performance for a particular application is difficult because it depends on many factors related to how memory is accessed, such as the percentage of reads and writes, opportunities for reuse (*i.e.*, reuse distances), and other aspects of the overall memory access pattern. For many applications, these factors could vary dramatically over each quantum of computation, which further complicates the issue. We claim, however, that our method will not have a dramatic *negative* effect on cache-miss rates.

5 Concluding Remarks

We have proposed a "spread-cognizant" scheduling method that decreases average and maximum spread when applied to both the PD^2 and EDF scheduling algorithms, and made certain guarantees about spread for a task set depending on the maximum weight task in the set. We then evaluated our scheduling method by showing its effect on raw parallelism and spread, as well as L2 cache-miss ratios in a simulated multicore environment.

There are many 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 explore support for critical sections and precedence constraints that incorporate our spread-cognizant scheduling policies. Third, we currently assume that all tasks to be co-scheduled have the same weight; we would like to remove this constraint. Finally, we would like to find other applications for which our scheduling policies provide a significant benefit, and perform experiments to show that benefit. Ultimately, we want to showcase our method by implementing applications on a real multicore system.

References

[1] J. Anderson, J. M. Calandrino, and U. Devi. Real-time scheduling on multicore platforms. To appear in *Proc. of the 12th IEEE Real-Time and Embedded Tech. and Apps. Symp.*, 2006.

[2] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. In *Proc. of the 13th Euromicro Conf. on Real-time Systems*, pages 76–85, 2001.

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

[4] E. Bampis, M. Caramia, J. Fiala, A. V. Fishkin, and A. Iovanella. Scheduling of independent dedicated multiprocessor tasks. *13th Annual Int'l Symp. on Algorithms and Computation*, 2002.

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

[6] U. Devi and J. Anderson. Tardiness bounds for global EDF scheduling on a multiprocessor. *Proc. of the 26th IEEE Real-time Systems Symp.*, 2005.

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

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

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

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

[11] 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.

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

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