Reducing Tardiness Under Global Scheduling by Splitting Jobs*

Jeremy P. Erickson and James H. Anderson
The University of North Carolina at Chapel Hill

Abstract

Under current analysis, soft real-time tardiness bounds applicable to global earliest-deadline-first scheduling and related policies depend on per-task worst-case execution times. By splitting job budgets to create subjobs with shorter periods and worst-case execution times, such bounds can be reduced to near zero for implicit-deadline sporadic task systems. However, doing so could potentially cause more preemptions and create problems for synchronization protocols. This paper analyzes this tradeoff between theory and practice by presenting an overhead-aware schedulability study pertaining to job splitting. In this study, real overhead data from a scheduler implementation in LITMUS^{RT} *was factored into schedulability analysis. This study shows that despite practical issues affecting job splitting, it can still yield substantial reductions in tardiness bounds for soft real-time systems.*

1 Introduction

For implicit-deadline sporadic task systems, a number of optimal multiprocessor hard real-time scheduling algorithms exist that avoid deadline misses in theory, as long as the system is not overutilized (e.g., [2, 13, 14]). However, all such algorithms cause jobs to experience frequent preemptions and migrations or are difficult to implement in practice.

For some applications, such as multimedia systems, some deadline tardiness is acceptable. For these applications, soft real-time scheduler options exist that have many of the advantages of optimal algorithms, without the associated practical concerns, at the cost of allowing some deadline misses. Such scheduler options include a wide variety of global algorithms that are reasonable to implement, do not give rise to excessive preemptions and migrations, and can ensure per-task tardiness bounds while allowing full platform utilization [12]. Such algorithms include the *global earliest deadline first* (*G-EDF*) scheduler and the improved *global fair lateness* (*G-FL*) scheduler [6, 7, 9, 10]. G-FL (see Sec. 2) is considered a *G-EDF-like* (*GEL*) scheduler, because under it, each job's priority is defined by a fixed point in time after its release, like under G-EDF.

Even if some amount of tardiness is tolerable in an application, it would still be desirable to have tardiness bounds that are reasonably small. In current tardiness analysis for GEL schedulers, tardiness bounds are expressed in terms of maximum job execution costs. If such bounds are deemed too large, then one potential solution is to split jobs into subjobs, which lowers executions costs, and hence tardiness bounds. However, job splitting increases the likelihood that the original job will be preempted/migrated frequently and thus can increase overheads that negatively impact schedulability. Also, as explained later, job splitting can cause problems for synchronization protocols. In this paper, we examine the practical viability of job splitting for reducing tardiness bounds under GEL schedulers in light of such complications.

Motivating example. For motivational purposes, we will repeatedly consider example schedules of a task system τ with three tasks, which we specify here using the notation (per-job execution cost, time between job releases): $\tau_1 = (4ms, 6ms)$, $\tau_2 = (9ms, 12ms)$, and $\tau_3 = (14ms, 24ms)$. Each job of τ_1, τ_2, and τ_3 has a deadline at the release time of its successor. An example G-EDF schedule for τ is given in Fig. 2, and an example G-FL schedule in Fig. 3. Observe that τ_3 misses a deadline at time 24 in Fig. 2. (This system meets the constraints for bounded tardiness in [6].)

A continuum of schedulers. In the implementation of job splitting we propose, all job splitting is done through budget tracking in the operating system (OS). That is, job splitting does not require actually breaking the executable code that defines a task into multiple pieces. We define the *split factor* of a task as the number of subjobs into which each of its jobs is split. With any GEL scheduler, existing tardiness bounds can be driven arbitrarily close to zero by arbitrarily increasing such split factors. In the "limit," i.e., when each subjob becomes one time unit (or quantum) in length, a GEL algorithm becomes similar in nature to algorithms within the Pfair family of optimal schedulers [2]. One

*Work supported by NSF grants CNS 1016954, CNS 1115284, and CNS 1239135; ARO grant W911NF-09-1-0535; and AFRL grant FA8750-11-1-0033.

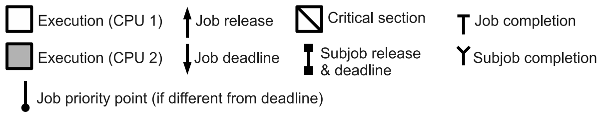

Figure 1: Key for schedules.

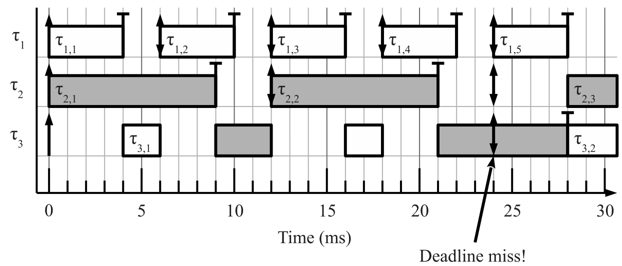

Figure 2: τ scheduled with G-EDF. The key in Fig. 1 is assumed in this and subsequent figures. (As explained in Sec. 2, $\tau_{i,j}$ denotes the j^{th} job of task τ_i.)

can thus view task split factors as tuning parameters that can be set to select a desired scheduler within a continuum of schedulers to achieve desired tardiness bounds. If tardiness were the only issue, then split factors would naturally be set arbitrarily high, but this raises practical concerns, as discussed earlier.

Returning to our example task system, Fig. 4 depicts a schedule for τ under G-EDF in which each job of τ_3 is split into two subjobs. Note that splitting is done in a way that preserves the task's original utilization. In this example, the tardiness of τ_3 is reduced by $1ms$, and no additional preemptions happen to be necessary.

Related work on job splitting. The idea of job splitting is not new and has been applied in other contexts. For example, job splitting has been proposed for reducing jitter in a control system in [5]. In that work, each job is split into three subjobs: the first reads data, the second performs necessary calculations, and the third outputs a control action. Priorities are selected to make the time between control actions as consistent as possible. Job splitting has also been proposed as a way to make rate-monotonic priorities reflect criticality [15]: job splitting can be applied to a critical task to reduce its period, and hence elevate its priority under rate-monotonic scheduling. The implementation of job splitting in RT-Linux on a uniprocessor has also been studied [17]. Additionally, a splitting technique similar to ours has been proposed to achieve mixed-criticality schedulability on a uniprocessor [16]. These are just a few examples concerning the use of job splitting that can be found in the literature.

Contributions. Our goal herein is to assess the practical usefulness of job splitting to reduce tardiness in GEL schedulers. In the first part of the paper (Secs. 4-6), we describe how to implement job splitting in the

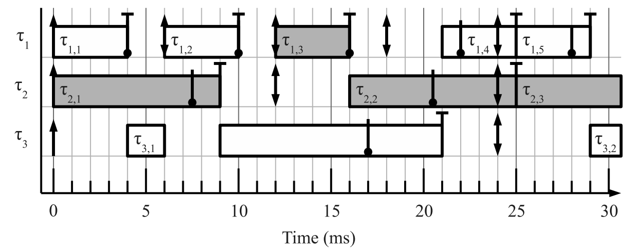

Figure 3: τ scheduled with G-FL.

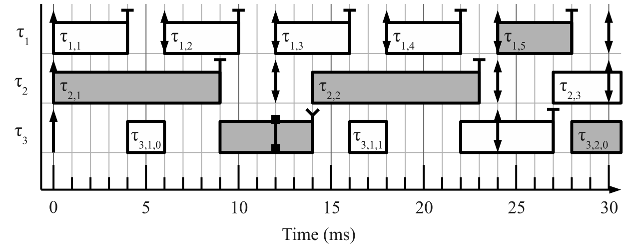

Figure 4: τ scheduled with G-EDF, where each job of τ_3 is split into two subjobs and jobs of the other tasks are not split.

OS by properly managing task budgets. We explain the needed budget management by first considering systems in which critical sections due to the use of synchronization protocols are not present, and by then discussing modifications that critical sections necessitate. Since the efficacy of job splitting depends on overheads, we also describe relevant overheads that must be considered and explain how they can be accounted for in schedulability analysis. For ease of exposition, we limit attention to G-EDF throughout this part of the paper because, within the class of GEL schedulers, G-EDF is the most widely understood algorithm.

In the second part of the paper (Sec. 7), we present an experimental evaluation of job splitting. In this part of the paper, we focus on G-FL because it (provably) has the best tardiness bounds of any GEL scheduler (using current analysis techniques) [7]. All of the results presented for G-EDF in the first part of the paper are easily adapted to apply to G-FL. In our evaluation, we utilize a new heuristic algorithm that automatically determines split factors. This algorithm is a contribution in itself as it frees programmers from having to specify split factors. We evaluate the usefulness of job splitting by comparing tardiness bounds that result with and without this heuristic algorithm applied (i.e., with job splitting and without it). In these experiments, real measured overheads from an implementation of G-FL in LITMUS$^{\text{RT}}$ [3] were applied and task systems both with and without critical sections were considered. The results of these experiments are quite striking. Our heuristic algorithm was found to often enable significant tardiness-bound reductions, even when a synchronization protocol is used. Reductions in the range 25% to 80% were quite common.

Before presenting the results summarized above, we first present needed background and discuss related work in more detail (Secs. 2–3).

2 Task Model

We consider a system τ of n *implicit deadline* sporadic tasks $\tau_i = (T_i, C_i)$ running on $m \geq 2$ processors, where T_i is the minimum separation time between subsequent releases of jobs of τ_i, and $C_i \leq T_i$ is the worst-case execution time of any job of τ_i. We denote the j^{th} job of τ_j as $\tau_{i,j}$. We assume that $n > m$. If this is not the case, then each task can be assigned its own processor, and each job of each τ_i will complete within C_i time units of its release. We assume that the OS enforces execution budgets, so that each job runs for at most C_i time units. We also assume that the relative deadline of each job equals its period. We use $U_i = C_i/T_i$ to denote the *utilization* of τ_i. All quantities are real-valued. We assume that

$$\sum_{\tau_i \in \tau} U_i \leq m, \tag{1}$$

which was demonstrated to be a necessary condition for the existence of tardiness bounds [12].

The focus of this work is the splitting of jobs into smaller subjobs with smaller periods and worst-case execution times, as depicted in Figs. 2–4. To distinguish between a task before splitting (e.g., τ_3 in Fig. 2) and the same task after splitting (τ_3 in Fig. 4), we define τ_i^{base} as the *base task* before splitting and τ_i^{split} as the *split task* after splitting. To disambiguate between base and split tasks, we also use superscripts on parameters: C^{base}, C^{split}, U^{base}, etc. A job of a base task is called a *base job*, while a split task is instead composed of *subjobs* of base jobs. We define the *split factor* of τ_i^{base}, denoted s_i, to be the number of subjobs per base job. In Fig. 4, $s_3 = 2$. The subjobs of a base job $\tau_{i,j}^{base}$ are denoted $\tau_{i,j,0}, \tau_{i,j,1}, \ldots, \tau_{i,j,s_i-1}$. $\tau_{i,j,0}$ is its *initial subjob* (e.g., the first subjob $\tau_{3,1,0}$ of $\tau_{3,1}^{base}$ in Fig. 4) and τ_{i,j,s_i-1} is its *final subjob* (e.g., the second subjob $\tau_{3,1,1}$ of $\tau_{3,1}^{base}$ in Fig. 4). The longest time that any job of τ_i^{base} waits for or holds a single outermost lock is denoted b_i. Split tasks use a variant of the sporadic task model that is described in Sec. 6, but the sporadic task model is assumed prior to Sec. 6.

If a job has absolute deadline d and completes execution at time t, then its *lateness* is $t - d$, and its *tardiness* is $\max\{0, t - d\}$. If such a job was released at time r, then its *response time* is $t - r$. We bound these quantities on a per-task basis, i.e., for each τ_i, we consider upper bounds on these quantities that apply to all jobs of τ_i. The *max-lateness* bound for τ is the largest lateness bound for any $\tau_i \in \tau$. Similarly, the *max-tardiness* bound for τ is the largest tardiness bound for any $\tau_i \in \tau$.

Let $\tau_{i,j}$ be a job of task τ_i released at time $r_{i,j}$. The *relative deadline* (of the task) is T_i, and the *absolute deadline* (of the job) is $r_{i,j} + T_i$. A scheduler is *G-EDF-like* (*GEL*) if the priority of $\tau_{i,j}$ is $r_{i,j} + Y_i$, where Y_i is constant across all jobs of τ_i. Y_i is referred to as the *relative priority point* (of the task) and $r_{i,j} + Y_i$ as the *absolute priority point* (of the job). Jobs with earlier absolute priority points have higher priority. G-EDF is the GEL scheduler with $Y_i = T_i$, and G-FL is the GEL scheduler with $Y_i = T_i - \frac{m-1}{m} C_i$ [7].

When a non-final subjob completes, the resulting change in deadline is a *deadline move* (*DLM*). In Fig. 4, a DLM occurs at time 14 for τ_3.

3 Prior Work

In this section, we discuss prior work that we utilize. In Sec. 3.1, we discuss work relating to tardiness bounds, and in Sec. 3.2, we discuss work relating to overhead analysis.

3.1 Tardiness Bounds

In this subsection, we briefly review relevant prior work for computing tardiness bounds. The purpose of this review is to show that prior tardiness bounds each approach zero as the maximum C_i in the system approaches zero. We will use the notation described in Sec. 2 rather than the original notation in the relevant papers. Tardiness bounds for G-EDF were originally considered by Devi and Anderson [6]. They define a value

$$x = \frac{\sum_{m-1 \text{ largest}} C_i - \min_{\tau_i \in \tau} C_i}{m - \sum_{m-2 \text{ largest}} U_i}$$

such that no task τ_i will have tardiness greater than $x + C_i$. An improved, but more complex, bound was later introduced [9]. While these works focused on G-EDF itself, later work proposed the new scheduler G-FL [7], with analysis similar to the improved analysis of G-EDF. G-FL usually provides a smaller maximum lateness bound for the task system. These improvements are based on analysis following the same basic proof structure as the original work [6], and they maintain the property that all tardiness bounds approach zero as the maximum C_i in the system approaches zero.

3.2 Overhead Analysis

In order to determine the schedulability of a task system in practice, it is necessary to determine relevant system overheads and to account for them in the analysis. We use standard methods proposed by Brandenburg [3] for G-EDF, as these methods can be applied to arbitrary GEL schedulers. Due to space constraints, we only provide here a brief overview of relevant overheads. For complete analysis, please consult [3].

Consider Fig. 5, which depicts a subset of the schedule in Fig. 2 with some additional overheads included.

1. From the time when an event triggering a release (e.g., a timer firing) occurs until the time that the corresponding interrupt is received by the OS, there is *event latency*, denoted ev (at time 18) in Fig. 5.

2. When the interrupt is handled, the scheduler must perform release accounting and may assign the released job to a CPU. This delay is referred to as *release overhead*, denoted rel (after time 18) in Fig. 5.

3. If the job is to be executed on a CPU other than the one that ran the scheduler, then an *interprocessor interrupt* (*IPI*) must be sent. In this case, the job will be delayed by the *IPI latency* of the system, denoted ipi (after time 18) in Fig. 5.

4. The scheduler within the OS must run when the IPI arrives (before time 19), creating *scheduling overhead*, denoted sch (before time 19) in Fig. 5.

5. After the scheduling decision is made, a context switch must be performed, creating *context switch overhead*, denoted cxs (at time 19) in Fig. 5.

Observe from Fig. 2 that $\tau_{3,1}$ had previously been preempted by $\tau_{1,3}$ at time 12. As a result of this earlier preemption, it experiences three additional costs when it is scheduled again after time 16.

1. When $\tau_{3,1}$ is scheduled again (time 16), it incurs another scheduling overhead sch and context switch overhead cxs.

2. Because $\tau_{3,1}$ was preempted, some of its cached data items and instructions may have been evicted from caches by the time it is scheduled again. As a result, $\tau_{3,1}$ will require extra execution time in order to repopulate caches. Although not depicted in Fig. 5, observe from Fig. 2 that $\tau_{3,1}$ is migrated to another processor at time 21, which may cause even greater cache effects. The added time to repopulate caches is called *cache-related preemption and migration delay* (*CPMD*) and is denoted cpd (before time 17) in Fig. 5.

Another overhead that occurs is the presence of interrupts, both from the periodic timer tick and from job releases. The maximum time for the timer tick interrupt service request routine is denoted tck in Fig. 5 (time 15), and the maximum cache-related delay from an interrupt is denoted cid in Fig. 5 (after time 15).

4 Split G-EDF Scheduling Algorithm

In this section, we describe the OS mechanisms necessary to implement job splitting under G-EDF. Although we will require the system designer to specify the split factor s_i for each job, we do not require the jobs to be split *a priori*. Instead, the OS will use the budget tracking schemes described in this section to perform DLMs at the appropriate times.

When certain events occur, the scheduler within the OS is called. We refer to this call as a *reschedule*. For example, a reschedule occurs whenever a job completes, so that another job can be selected for execution. In our implementation of splitting in LITMUS$^{\mathrm{RT}}$, part of the scheduling process involves checking whether the currently executing job needs a DLM and to perform the DLM if so.

In this section, we let $C_i^{split} = C_i^{base}/s_i$ and $T_i^{split} = T_i^{base}/s_i$. For example, in Fig. 4, $C_3^{split} = 14/2 = 7$ and $T_3^{split} = 24/2 = 12$.

The tardiness analysis reviewed in Sec. 3.1 continues to hold if jobs become available for execution before their release times, as long as their deadlines are based on release times that follow the minimum separation constraint of the sporadic task model. The technique of allowing jobs to run before their release times is called *early releasing* [1]. Allowing subjobs to be released early may prevent tasks from suspending unnecessarily and allows us to alter the deadline of a job $\tau_{i,j}$ only when it has executed for a multiple of C_i^{split}. With early releasing, we do not have to consider the wall clock time when determining a split job's deadline, because we can instead consider its cumulative execution time. Additionally, as discussed in Sec. 5, early releasing prevents the same job from having to incur certain overheads multiple times.

We will track the budget of each $\tau_{i,j}$ in order to simulate the execution of $\tau_{i,j,0}, \tau_{i,j,1}, \ldots \tau_{i,j,s_i}$ with T_i^{split} time units between each pair of subjob releases and with each subjob executing for up to C_i^{split} time units. In order to do so, we define several functions below with respect to time. These functions are only defined for time t such that τ_i^{base} has a job that is ready for execution (it is released and its predecessor job has completed) but has not completed. We let $J_i(t)$ denote this job. For example, in Fig. 4, $J_3(t)$ denotes $\tau_{3,1}^{base}$ for any $t \in [0, 27)$, and $\tau_{3,2}^{base}$ after $t = 27$. Several of these func-

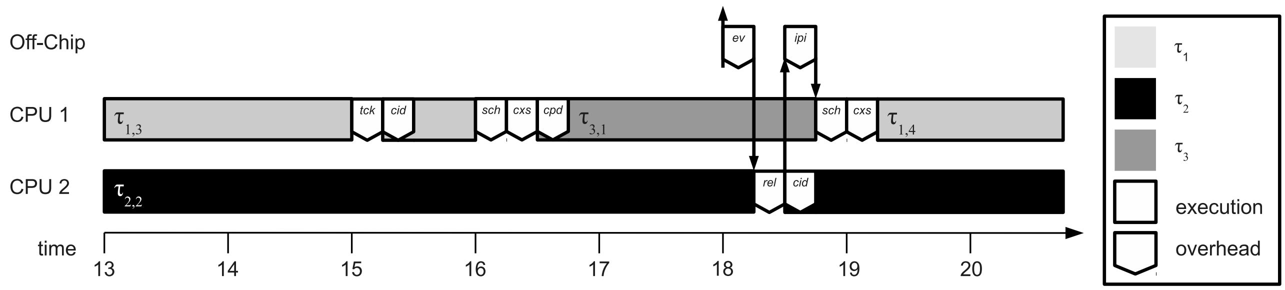

Figure 5: A subset of the schedule from Fig. 2 with overheads included. The execution times have been slightly reduced to make room for overheads.

tions are explicitly labelled as "ideal" functions that ignore critical sections—deviation from "ideal" behavior due to critical sections will be described in Sec. 6.

- The *current execution* $e_i(t)$ is the amount of execution that $J_i(t)$ has already completed before time t. In Fig. 4, $e_3(4) = 0$ and $e_3(5) = 1$. Our definition of $e_i(t)$ allows us to keep track of how many subjobs of $J_i(t)$ have already completed.

- The *current release* $r_i(t)$ is the release time of $J_i(t)$. Note that $r_i(t)$ is the release time of the current base job, not the current subjob. In Fig. 4, $r_3(4) = r_3(17) = 0$ and $r_3(29) = 24$.

- The *ideal subjob* $j_i(t)$ is the index of the subjob of $J_i(t)$ that should be executing at time t, ignoring the effect of critical sections. In other words, it is the index of the subjob that should be executing based on the number of multiples of C_i^{split} that $J_i(t)$ has completed by time t. It is defined as follows:
$$j_i(t) = \left\lfloor \frac{s_i \cdot e_i(t)}{C_i^{base}} \right\rfloor. \qquad (2)$$
In Fig. 4, $j_3(4) = 0$, $j_3(17) = 1$, and $j_3(29) = 0$. (Recall that subjobs are zero-indexed.)

- The *ideal next DLM* $v_i(t)$ is the time for the next DLM after time t, ignoring the effect of critical sections and assuming that $J_i(t)$ is scheduled continuously from time t until $v_i(t)$. In other words, $v_i(t)$ is the time when the current ideal subjob should end assuming that it is not preempted. It is defined as follows:
$$v_i(t) = t + (j_i(t) + 1)C_i^{base}/s_i - e_i(t). \qquad (3)$$
In Fig. 4, $v_3(4) = 11$. Observe that, because $\tau_{3,1,0}$ is actually preempted at time 6, the DLM actually does not occur until time 14.

- The *ideal subjob release* $\rho_i(t)$ is the release time for the current ideal subjob. It is defined as follows:
$$\rho_i(t) = r_i(t) + T_i^{split} j_i(t). \qquad (4)$$
(4) reflects that the subjobs are released every T_i^{split} time units, and the first subjob is released at the same time as the corresponding base job. In Fig. 4, $\rho_3(4) = 0$. Although it does not occur in Fig. 4, it is possible (due to early releasing) for the ideal subjob release to be after that subjob actually commences execution.

- The *ideal deadline* $d_i(t)$ is the deadline that should be active for $J_i(t)$ at time t, ignoring the effect of critical sections. In other words, it is the deadline of the ideal subjob $j_i(t)$. It is defined as follows:
$$d_i(t) = \rho_i(t) + T_i^{split} \qquad (5)$$
(5) follows from the definition of G-EDF scheduling. In Fig. 4, $d_3(4) = 12$.

- The *current deadline* $\delta_i(t)$ is the deadline that the scheduler actually uses for $J_i(t)$ at time t. This value is maintained by the budget tracking algorithm we describe in this section, rather than being merely a definition like the functions above. Because there are no critical sections in Fig. 4 (as we are assuming in this section), $\delta_i(t)$ should be $d_i(t)$ for all i and all t. Therefore, $\delta_3(4)$ should be 12.

With these definitions in place, we define budget tracking rules in order to maintain the invariant $\delta_i(t) = d_i(t)$.

- **R1.** If a job of τ_i^{base} is released at time t, then $\delta_i(t)$ is assigned to $d_i(t)$.

  In Fig. 4, applying this rule at time 0, we have $\delta_3(0) = 12$.

- **R2.** Whenever a non-final subjob of τ_i^{base} is scheduled at time t to run on a CPU, a *DLM timer* is assigned to force a reschedule on that CPU at time $v_i(t)$. Whenever τ_i^{base} is preempted, the DLM timer is cancelled.

  In the schedule depicted in Fig. 4, the DLM timer for τ_3 is set at time 4 to fire at time $v_3(4) = 11$.

However, the DLM timer is cancelled at time 6 when τ_3 is preempted. When τ_3 is selected for execution again at time 9, the DLM is set to fire at time 14. It does fire at that time and forces a reschedule. Because only the final subjob remains, the timer is not set at time 16.

- **R3.** If the scheduler is called at time t on a CPU that was running τ_i^{base}, then $\delta_i(t)$ is assigned the value $d_i(t)$, potentially causing a DLM.

  In Fig. 4, the scheduler is called several times on a CPU that was running τ_3^{base}, including at times 6 and 14. At time 6, $d_3(t) = \delta_3(t)$ already held, so a DLM does not occur. However, a DLM occurs at time 14 because $d_3(t) > \delta_3(t)$ is established, causing $\delta_3(t)$ to be updated.

5 Overhead Analysis

We now describe how to implement job splitting in an efficient manner and how the overheads from our implementation differ from those in Sec. 3.2. Critical sections are not considered until Sec. 6. An illustration of overheads due to job splitting is given in Fig. 6.

Whenever a DLM is necessary by Rule R3 above, the scheduler can simulate a job completion for the old subjob (with the old deadline), followed by an immediate arrival for the new subjob (with the new deadline). The same situation occurs when a tardy job completes after the release of its successor. Therefore, the same accounting can be used for both the case where the DLM timer fires, ending a subjob on its processor, and for the case of a normal job completion.

Having considered the direct overheads produced by DLMs, we now consider other relevant overheads that happen while running the system. As a simple overhead accounting method, we can simply analyze split tasks rather than base tasks, treating subjobs as jobs. In so doing, it is necessary to account for the cache-related delays that can be caused by the preemption of the base job between subjobs, because this preemption is not necessarily caused by a new release of another job (or subjob). Treating each subjob as a base job is actually more pessimistic than necessary. When the release timer fires on behalf of a task, the time spent processing the resulting interrupt may delay part of the execution of a different task. However, the time spent processing the DLM timer interrupt will not delay any other task. (If a different task is selected after the DLM, that case does not differ from a normal job completion, as discussed above.) For example, when the system in Fig. 4 is executed, a release interrupt for $\tau_{3,1}$ will only occur at time 0, not at time 12 or time 14.

In addition, each non-initial subjob becomes available immediately when its predecessor completes. Because the scheduler in LITMUS$^{\mathrm{RT}}$does not release the global scheduler lock between processing a job completion and the next arrival, if the new subjob has sufficient priority to execute, then it will run on the same CPU as its predecessor. There are two improvements that are made possible by this observation. First, only the initial subjob of each base job can experience event latency or require an IPI. Second, only the initial subjob of each base job can preempt another job and thus cause preemption-related overheads.

6 Handling Critical Sections

One of the advantages of GEL schedulers is that they are *job-level static priority* (*JLSP*) algorithms, which is important for synchronization mechanisms [3]. However, when splitting is introduced, a GEL algorithm is no longer truly JLSP. If a subjob ends while waiting for or holding a lock, then the priority of the underlying job is changed, potentially violating the assumptions of synchronization algorithms. Furthermore, if a locking protocol operates nonpreemptively, then it is not possible to split a job while it is waiting for or holding a critical section. Fortunately, we can solve both problems by simply extending subjob budgets for as long as a resource request is active. A similar technique was proposed previously for aperiodic servers [11].

In order to support the necessary budget extensions, we use a more complicated set of rules than those described in Sec. 4. To illustrate the behavior of our modified algorithm, we present in Fig. 7 a modification of the schedule from Fig. 4 with the addition of critical sections. Our new rules allow the budget for a subjob to be extended when its DLM is delayed. Furthermore, because this delay does not change the expressions for $j_i(t)$, $v_i(t)$, $\rho_i(t)$, or $d_i(t)$, the next subjob implicitly has its budget shortened. Essentially, we are only allowing each DLM to "lag" behind the ideal DLM by at most b_i units of the corresponding base job's execution. It is even possible for a subjob to be implicitly skipped by this mechanism if $b_i > C_i^{split}$.

- **R1.** If a job of τ_i^{base} is released at time t, then $\delta_i(t)$ is assigned to $d_i(t)$.

  This rule is identical to Rule R1 from Sec. 4.

- **R2.** Whenever a non-final subjob of τ_i^{base} is scheduled at time t to run on a CPU, a *DLM timer* is assigned to force a reschedule on that CPU at time $v_i(t)$. Whenever τ_i^{base} is preempted, or τ_i^{base} requests a resource, the DLM timer is cancelled.

  In the schedule in Fig. 7, the DLM timer for τ_3 is set at time 9 to fire at time 14, but is cancelled at time 13 when τ_3^{base} requests a resource. Because

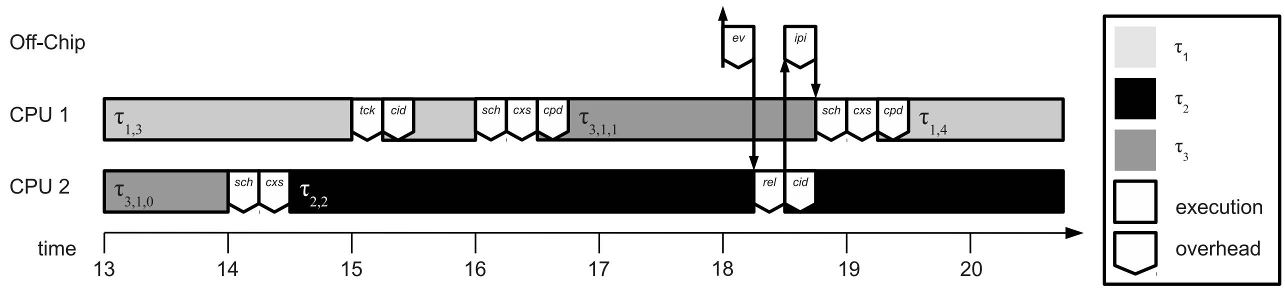

Figure 6: A subset of the schedule from Fig. 4 with some overheads included.

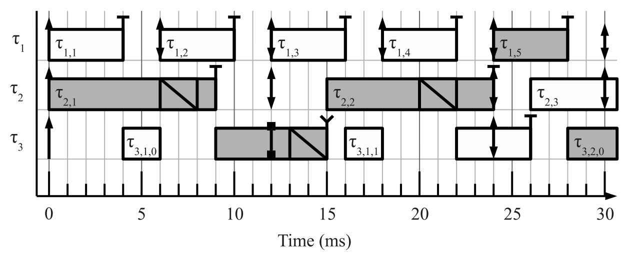

Figure 7: τ scheduled with G-EDF, where τ_3 is split into two subjobs, the other tasks are not split, and critical sections are present.

only a final subjob remains after time 15, however, the timer will not be set again.

- **R3.** Whenever a critical section ends, if $d_i(t) > \delta_i(t)$, then a reschedule is forced.

  Observe that for $t \in [14, 15)$ in Fig. 7, the current subjob of $\tau_{3,1}$ ($\tau_{3,1,0}$) is an earlier subjob than that indexed by $j_3(t)$. Thus, when the critical section ends, a DLM should occur. Triggering a reschedule will cause the needed DLM.

- **R4.** If the scheduler is called at time t on a CPU that was running τ_i^{base}, and τ_i^{base} is neither waiting for nor holding a resource, then $\delta_i(t)$ is assigned the value $d_i(t)$, potentially causing a DLM.

  This rule is nearly identical to Rule R3 in Sec. 4 and functions the same way, except in the case that the scheduler is invoked due to a job release from another task while τ_i^{base} is waiting for or holding a resource. However, if a DLM does occur, then the scheduler could have been invoked either due to Rule R2 or Rule R3 as modified above. In Fig. 7 it is invoked due to Rule R3 at time 15.

We let C_i^{split} denote the ideal worst-case execution time of a subjob, ignoring critical sections. When we account for critical sections, a single subjob of a job from τ_i can run for as long as $C_i^{split} + b_i$. Nonetheless, τ_i's processor share over the long term is not affected, because the total execution of all subjobs of a base job must equal the execution of that base job. Prior tardiness analysis [7] is described in a modified form accounting for critical sections in an online appendix [8]. The resulting tardiness bounds are increased by approximately b_i with no utilization loss.

7 Experiments

As stated in Sec. 1, we discussed G-EDF in prior sections only for ease of exposition. In this section, we instead consider G-FL, which has better maximum lateness bounds. Prior work [3] has shown that, when the effects of overheads are considered, the best scheduling strategy for bounded tardiness is *clustered* scheduling, where CPUs are grouped along cache boundaries and a global scheduling algorithm is used within each cluster. We thus consider *clustered fair lateness* (*C-FL*), where G-FL is used within each cluster.

Our experiments consisted of two phases. In the first phase, we measured overheads from an actual implementation of C-FL. In the second phase, we used these measurements as part of an overhead-aware schedulability study. This two-phase methodology has been used in several prior studies (e.g., see [3]).

First phase: overhead measurement We performed our overhead measurements on a 24-core Xeon L7455 (2.13GHz) system with 64GB of RAM. On that system, pairs of cores share an L2 cache and the cores on each six-core chip share an L3 cache. We considered four variants of C-FL. Under C-FL-L2, CPUs are grouped at the level of L2 caches, and all processors both receive interrupts and execute tasks. Under C-FL-L2-RM, CPUs are still grouped at the level of L2 caches, but one CPU operates as a *release master* (*RM*) that receives all interrupts but does not execute tasks. C-FL-L3 and C-FL-L3-RM are analogous to C-FL-L2 and C-FL-L2-RM, respectively, but group CPUs according to L3 caches rather than L2 caches.

For the first phase of our experiments, we implemented all four C-FL variants with splitting and measured relevant overheads in a manner similar to the overhead measurements in [3]. Recall that the relevant overheads are described in Sec. 3.2.

Second phase: schedulability study For the second phase of our experiments, we performed a schedulability study considering the same 24-core platform considered in the first phase.

We assessed schedulability trends by generating implicit-deadline task sets based on the experimental design from [3]. We generated task utilizations using either a uniform, a bimodal, or an exponential distribution. For task sets with uniformly distributed utilizations, we used either a *light* distribution with values randomly chosen from $[0.001, 0.1]$, a *medium* distribution with values randomly chosen from $[0.1, 0.4]$, or a *heavy* distribution with values randomly chosen from $[0.5, 0.9]$. For tasks sets with bimodally distributed utilizations, values were chosen uniformly from either $[0.001, 0.5]$ or $[0.5, 0.9]$, with respective probabilities of 8/9 and 1/9 for *light* distributions, 6/9 and 3/9 for *medium* distributions, and 4/9 and 5/9 for *heavy* distributions. For task sets with exponentially distributed utilizations, we used exponential distributions with a mean of 0.10 for *light* distributions, 0.25 for *medium* distributions, or 0.50 for *heavy* distributions. Utilizations were drawn until one was generated between 0 and 1. We generated integral task periods using a uniform distribution from $[3ms, 33ms]$ for *short* periods, $[10ms, 100ms]$ for *moderate* periods, or $[50ms, 250ms]$ for *long* periods.

When assessing schedulability in the presence of locking, critical sections were chosen uniformly from either $[1\mu s, 15\mu s]$ for *short* critical sections, $[1\mu s, 100\mu s]$ for *medium* critical sections, or $[5\mu s, 1280\mu s]$ for *long* critical sections. We varied the lock contention via two parameters, n_r, which is the number of resources, and p_{acc}, which is the probability that any task accesses a given resource. We performed tests with $n_r = 6$ and $n_r = 12$, and with $p_{acc} = 0.1$ and $p_{acc} = 0.25$. For a task using a given resource, we generated the number of accesses uniformly from the set $\{1, 2, 3, 4, 5\}$. These parameter choices are a subset of those used by Brandenburg [3] because, unlike Brandenburg, we opted to perform experiments on a larger variety of working set sizes (see below) to facilitate better comparisons with experiments without locking. A prior implementation study [4] demonstrated that for typical soft real-time applications, the majority of critical sections are less than $10\mu s$. Therefore, the short critical section distribution is likely to be the most common in many settings.

For each tested set of distribution parameters, we generated 100 task sets for each utilization cap of the form $\frac{24i}{20}$ where i is an integer in $[1, 20]$, in order to get a reasonable spread of utilization caps. Tasks were generated until one was created that would cause the system to exceed the given utilization cap. That task was then discarded. We tested each task set for each choice of working set size (WSS) within $\{2^2KB, 2^3KB, \ldots, 2^{11}KB\}$. The WSS for a base job is defined as its total memory footprint, and the WSS for each subjob was assumed to be the same as that of its base job. For tests not involving locking, we considered each combination of task set and WSS under each of the four C-FL variants. For tests involving locking, we instead considered the two C-FL variants using a release master, and we assumed that the mutex queue spinlock locking protocol [3] was used.

For each combination of task set and parameters, we first assigned the tasks to clusters using a worst-fit decreasing heuristic: we ordered tasks by decreasing utilization, and we placed each task in order on the CPU with the most remaining capacity. This heuristic was intended to maximize the available utilization for overheads. We then evaluated the tardiness without splitting by first inflating task parameters using methods from [3], and then using the inflated task system in the tardiness bound computation method in [8]. We ignored task sets that were either not schedulable under C-FL (without splitting) or resulted in zero tardiness, because our goal was to show improvements upon previously available schedulers. The number of ignored task sets varied greatly with respect to the utilization distribution, period distribution, and utilization cap. For example, very large utilization caps usually led to unschedulable systems once overheads were accounted for, even without splitting.

For each considered task set, we applied task splitting using the heuristic described below. When computing tardiness bounds after splitting, we first inflated the task parameters for overheads as before, but the method for inflating parameters was modified according to Sec. 5. We compared the maximum tardiness bounds for the inflated task sets before and after splitting. (Because $s_i = 1$ is allowed by our algorithm, every considered task set is scheduable under C-FL with splitting by definition.)

Heuristic for determining s_i In order to use splitting to reduce tardiness bounds, it is necessary to determine appropriate s_i values. To do so, we used a simple heuristic algorithm, described in Algos. 1–2. Due to space constraints, we do not depict the function `comp_max_lateness()`, which accounts for overheads and returns the maximum lateness for the entire task system if it is schedulable and $\perp$ if it is not. `comp_max_lateness()` uses the techniques discussed above from [3, 8] and Sec. 5. Similarly, we do not depict the function `split_candidates()`, which returns a list of tasks in a cluster ordered by contribution to the lateness bounds. `split_candidates()` is based on the details of the lateness bound computation in [8].

```
Input: Task system τ, cluster c, old maximum
       lateness old_max_late
Output: The new maximum lateness after
        splitting, or ⊥ if no improvement is
        possible
1  split_candidates := get_split_candidates(τ, c)
2  for τ_i ∈ split_candidates do
3      if not split_saturated_i then
4          s_i := s_i + 1
5          max_late := comp_max_lateness(τ)
6          if max_late = ⊥ then
7              s_i := s_i − 1
8              split_saturated_i := true
9          else if max_late < old_max_late then
10             return max_late
11         else
12             s_i := s_i − 1
13 return ⊥
```

Algorithm 1: Function `try_one_split`

```
Input: Task system τ
Output: The new maximum lateness after
        splitting
1  max_late := comp_max_lateness(τ)
2  keep_splitting := true
3  while keep_splitting do
4      keep_splitting := false
5      clust_list := clusters sorted by decreasing
                     maximum lateness
6      for c ∈ clust_list do
7          new_late := try_one_split(τ, c, max_late)
8          if new_late ≠ ⊥ then
9              max_late := new_late
10             keep_splitting := true
11 return max_late
```

Algorithm 2: Splitting heuristic

The function `try_one_split` (Algo. 1) attempts to reduce the lateness bound for the system by increasing a single s_i by 1. It may make multiple attempts before finding a splitting that actually reduces the maximum lateness bound for the system. For each task τ_i attempted, the algorithm first increases s_i in line 4. If this increase improves the maximum lateness bound, the function returns in line 10, so no other s_i values will be affected. Otherwise, if the increase either results in an unschedulable system or simply no improvement to the maximum lateness bound, it is reverted (lines 7 and 12). To improve efficiency, `try_one_split` uses the per-task variable $\mathsf{split\_saturated}_i$, which is initially assumed to be **false**, but which is set to **true** in line 8, when increasing s_i by 1 would result in an unschedulable system. The value of $\mathsf{split\_saturated}_i$ is assumed to persist between calls to `try_one_split`. The purpose of accounting for $\mathsf{split\_saturated}_i$ is to avoid repeatedly trying to further split τ_i.

Algo. 2 computes values for s_i for all τ_i. It consists of a loop (lines 3–10) that is run repeatedly until the maximum lateness bound has stopped improving. Within this loop, the algorithm calls `try_one_split` on each cluster, ordered by the largest to the smallest maximum lateness bound. The purpose of calling `try_one_split` on multiple clusters is that in a system with locking, reducing the lateness bound in one cluster may reduce the lateness bounds for other clusters. Thus, even if there is no task to split in the cluster with the largest maximum lateness bound, it may be possible to reduce the lateness bound for that cluster by reducing the lateness bounds in a different cluster.

Results. Examples of results without locking are depicted in Figs. 8 and 9, which have the same key and plot average maximum tardiness bounds as a function of WSS. (Additional results can be found in the online appendix [8]. In total, our experiments resulted in several hundred graphs.) Observe that improvements over 25% are common, and can be nearly 100% in some cases. Because task systems with higher WSSs are more likely to be unschedulable even without splitting, higher WSSs often represent significantly smaller groups of task sets and are skewed towards task sets with smaller utilizations. This can cause a nonincreasing trend in the tardiness bounds with increased WSS for C-FL, but our purpose is to compare the effect of splitting when bounded tardiness is already achievable by C-FL. An overall trend from our experiments is that splitting provides more benefit when jobs are longer (larger utilizations and longer periods). This phenomenon occurs because the additional overheads from splitting are proportional to the split factor rather than job length, so the additional overheads are relatively smaller in comparison to longer jobs.

Fig. 10 has the same key as Figs. 8 and 9, but plots the average maximum tardiness bound as a function of the system utilization cap rather than WSS. (C-FL-L2-RM was not able to schedule any systems in this particular case.) Observe that the bounds with splitting (dashed lines) tend to grow more slowly than the bounds without splitting (solid lines) until they grow drastically before all tested task sets are unschedulable. This phenomenon occurs because the overheads from splitting use some of the system's remaining utilization, and when very little utilization is available tasks cannot be split as finely.

Figs. 11–13 share a key (distinct from that of Figs. 8–10) and depict the behavior of the system in

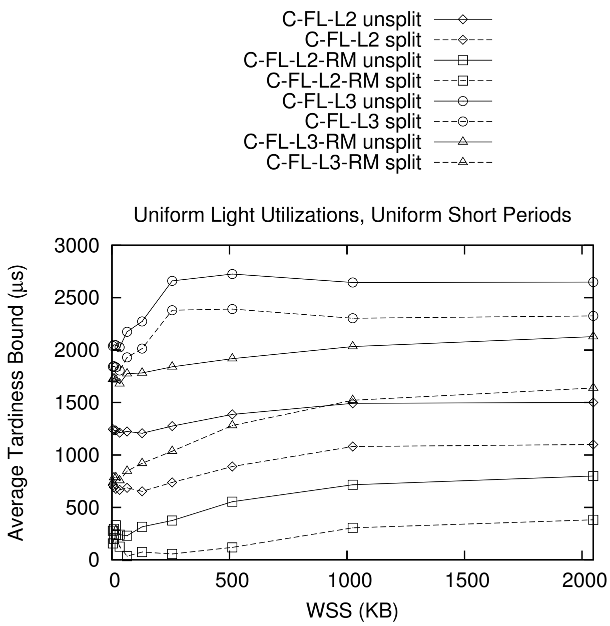

Figure 8: Light Uniform Utlization, Short Uniform Periods.

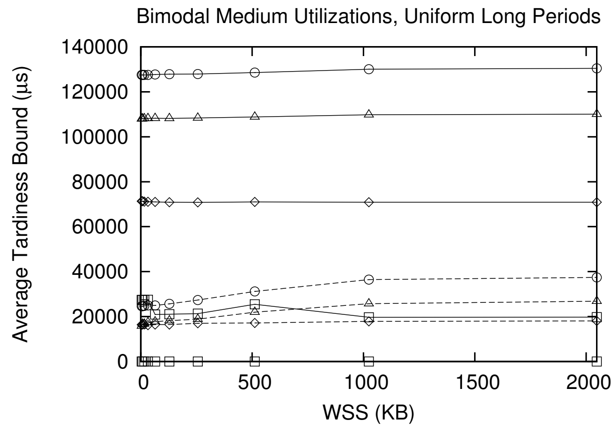

Figure 9: Medium Bimodal Utilization, Long Uniform Periods.

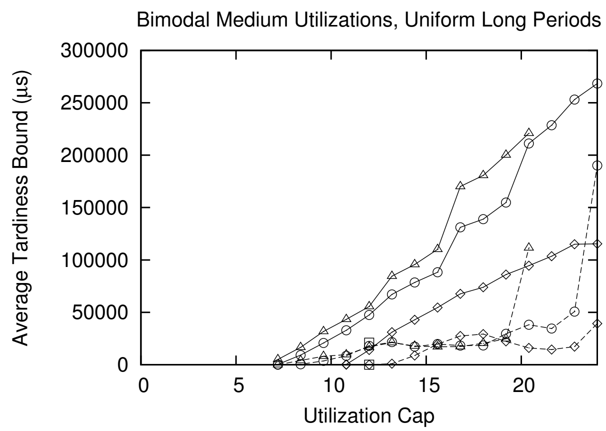

Figure 10: Medium Bimodal Utilization, Long Uniform Periods, WSS = 128KB. This graph is with respect to utilization cap instead of WSS.

the presence of locks. In order to compute blocking terms, we pessimistically assumed that every critical section in a base job could be repeated by each of its subjobs. Observe that significant gains from splitting are available most of the time despite this pessimism. In several cases, the improvement is on a similar order of magnitude to the gains without locking. However, for C-FL-L3-RM with long locks, there is no benefit to splitting, so the "split" and "unsplit" lines overlap. This phenomenon occurs because when the critical sections are long, the extra blocking time from multiple subjobs can quickly overutilize the system. These results demonstrate that when critical sections are not too long, job splitting is likely to be a useful technique even in the presence of locks.

8 Conclusions

Tardiness bounds established previously for GEL schedulers can be lowered in theory by splitting jobs. However, such splitting can increase overheads and create problems for locking protocols. In this paper, we showed how to incorporate splitting-related costs into overhead analysis and how to address locking-related concerns. We then applied these results in a schedulablity study in which real measured overheads were considered. This study suggests that job splitting can viably lower tardiness bounds in practice.

As future work, we could also examine how to account for suspension-based locking protocols. The strategy used in this paper should also be applicable with few or no modifications when using *suspension oblivious* [3] analysis that treats suspensions as computation. However, it could also be extended for use with *suspension aware* [3] analysis that explicitly accounts for suspensions.

References

[1] J. H. Anderson and A. Srinivasan. Early-release fair scheduling. In *ECRTS*, pages 35–43, 2000.

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

[3] B. B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

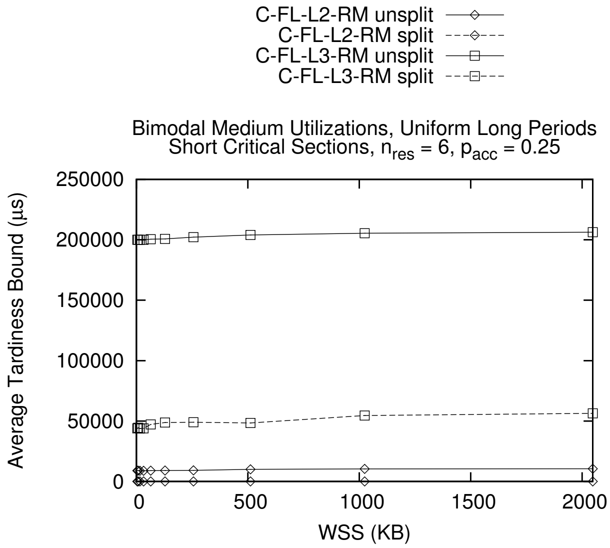

Figure 11: Medium Bimodal Utilization, Long Uniform Periods, Short Critical Sections, $n_r = 6$, $p_{acc} = 0.25$.

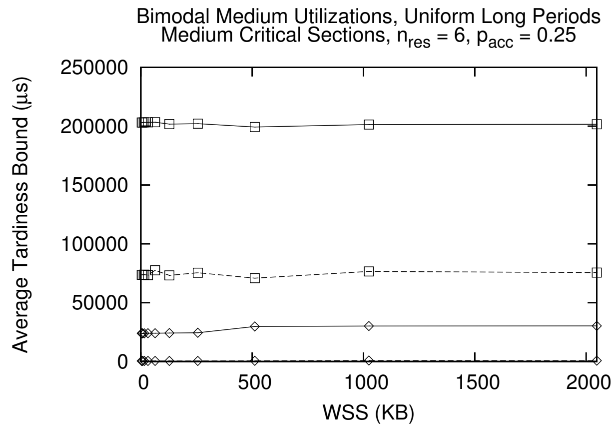

Figure 12: Medium Bimodal Utilization, Long Uniform Periods, Medium Critical Sections, $n_r = 6$, $p_{acc} = 0.25$.

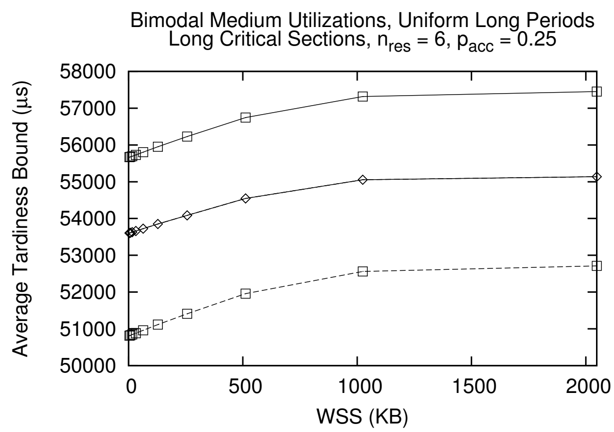

Figure 13: Medium Bimodal Utilization, Medium Uniform Periods, Long Critical Sections, $n_r = 6$, $p_{acc} = 0.25$.

[4] B. B. Brandenburg and J. H. Anderson. Feather-trace: A light-weight event tracing toolkit. In *OSPERT*, pages 61–70, 2007.

[5] A. Crespo, I. Ripoll, and P. Albertos. Reducing delays in rt control: the control action interval. In *IFAC World Congress*, 1999.

[6] U. C. Devi and J. H. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. *Real-Time Sys.*, 38(2):133–189, 2008.

[7] J. P. Erickson and J. H. Anderson. Fair lateness scheduling: Reducing maximum lateness in G-EDF-like scheduling. In *ECRTS*, pages 3–12, 2012.

[8] J. P. Erickson and J. H. Anderson. Appendix to reducing tardiness under global scheduling by splitting jobs. `http://cs.unc.edu/~anderson/papers.html`, April 2013.

[9] J. P. Erickson, U. Devi, and S. K. Baruah. Improved tardiness bounds for global EDF. In *ECRTS*, pages 14–23, 2010.

[10] J. P. Erickson, N. Guan, and S. K. Baruah. Tardiness bounds for global EDF with deadlines different from periods. In *OPODIS*, pages 286–301, 2010. Revised version at `http://cs.unc.edu/~jerickso/opodis2010-tardiness.pdf`.

[11] T.M. Ghazalie and T. P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Sys.*, 9:31–67, 1995.

[12] H. Leontyev and J. H. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. *Real-Time Sys.*, 44(1):26–71, 2010.

[13] T. Megel, R. Sirdey, and V. David. Minimizing task preemptions and migrations in multiprocessor optimal real-time schedules. In *RTSS*, pages 37–46, 2010.

[14] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt. RUN: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In *RTSS*, pages 104–115, 2011.

[15] L. Sha and J.B. Goodenough. Real-time scheduling theory and ada. *Computer*, 23(4):53–62, April 1990.

[16] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga. Mixed critical earliest deadline first. Technical Report TR-2012-22, Verimag Research Report, 2012.

[17] J. Vidal, A. Crespo, and P. Balbastre. Task decomposition implementation in rt-linux. In *15th IFAC World Congress on Automatic Control.*, pages 944–949. Elsevier Science, 2002.