ABSTRACT

Semi-partitioned scheduling is an approach to multiprocessor real-time scheduling where most tasks are fixed to processors, while a small subset of tasks is allowed to migrate. This approach offers reduced overhead compared to global scheduling, and can reduce processor capacity loss compared to partitioned scheduling. Prior work has resulted in a number of semi-partitioned scheduling algorithms, but their correctness typically hinges on a complex intertwining of offline task assignment and online execution. This brittleness has resulted in few proposed semi-partitioned scheduling algorithms that support dynamic task systems, where tasks may join or leave the system at runtime, and few that are optimal in any sense. This paper introduces EDF-sc, the first semi-partitioned scheduling algorithm that is optimal for scheduling (static) soft real-time (SRT) sporadic task systems and allows tasks to dynamically join and leave. The SRT notion of optimality provided by EDF-sc requires deadline tardiness to be bounded for any task system that does not cause over-utilization. In the event that all tasks can be assigned as fixed, EDF-sc behaves exactly as partitioned EDF. Heuristics are provided that give EDF-sc the novel ability to stabilize the workload to approach the partitioned case as tasks join and leave the system.

CCS CONCEPTS

• Computer systems organization → Real-time systems.

KEYWORDS

multicore processors, real-time, semi-partitioned scheduling, dynamic, reweighting

1 INTRODUCTION

Semi-partitioned scheduling is a compromise between the traditional global and partitioned approaches to multiprocessor scheduling, where most tasks are assigned as fixed, or only able to execute on one processor, while a small subset of tasks are migrating, or able to execute on more than one processor. This approach gives reduced overhead compared to global scheduling because few tasks can migrate, and can avoid the capacity loss of up to 50% inherent to partitioned scheduling. Semi-partitioned scheduling was first proposed for soft real-time (SRT) scheduling [1], where deadline misses are acceptable as long as they are bounded in length. In this paper, we focus our attention on this type of SRT system.

Unfortunately, most proposed semi-partitioned scheduling algorithms depend on an inflexible offline task-assignment phase to provide correctness guarantees. This offline assignment is usually quite brittle, and is intertwined with the scheduling rules used online so that the assignment cannot be changed at runtime, at least not without incurring prohibitive computational costs. This precludes the possibility of supporting dynamic task systems, in which tasks may be added to or removed from the system at runtime. In this paper, we alleviate this restriction by introducing a simple SRT semi-partitioned scheduling algorithm that is optimal for (static) sporadic task systems, and that supports dynamic task systems.

Prior work. The first semi-partitioned scheduling algorithm to be proposed was EDF-fin [1], which guarantees bounded tardiness with no overall system utilization cap (beyond the obvious cap of m on an m-processor system), but is not optimal because it requires that each task has utilization at most 1/2. Since then, numerous other semi-partitioned scheduling algorithms have been proposed [2–4, 7–9, 12–14, 17, 19–21, 24–26, 30–32]. Of these, two are especially relevant to this work. Anderson et al. [2] proposed EDF-os, the first SRT optimal semi-partitioned scheduling algorithm. Unfortunately, EDF-os does not support dynamic task systems because its tardiness bounds critically hinge on properties established during its offline task-assignment phase, and these properties are difficult to maintain if changes to the system occur. Casini et al. [14] proposed an approximate C=D splitting algorithm, which is to our knowledge the only prior semi-partitioned scheduling algorithm that supports dynamic workloads. Unlike our work, however, Casini et al. focused on hard real-time systems, for which their algorithm is not optimal.

Contributions. In this paper, we present the first semi-partitioned scheduling algorithm that is optimal in the SRT case, and that supports dynamic task systems. This algorithm, called EDF-sc (earliest-deadline-first-based semi-partitioned scheduling with containers), takes a novel yet simple approach to semi-partitioned scheduling. Rather than performing an inflexible offline assignment of tasks
to processors and then scheduling tasks by a set of rules based on this assignment, EDF-sc schedules migrating tasks on a global EDF (GEDF) basis alongside a set of containers that are each assigned to a unique processor. Each container holds the set of fixed tasks on its corresponding processor. When a container is selected to run, it schedules its fixed tasks using uniprocessor EDF. This simple, hierarchical approach gives bounded tardiness for all tasks, and to the best of our knowledge is the first algorithm to do so irrespective of the assignment of tasks to processors. Because of this property, we are able to provide rules to add and remove tasks from the system at runtime. We introduce a number of heuristics for adding and removing tasks that stabilize the workload by reducing the number of migrating tasks over time. Finally, we present the results of a migration-overhead-aware schedulability study comparing EDF-sc to GEDF in static systems. We also show the results of an experimental evaluation of our heuristics, and compare tardiness under EDF-sc to that under GEDF. These experiments show that EDF-sc generally provides higher schedulability than GEDF, and can give reduced tardiness compared to GEDF for systems with low-utilization tasks. They also show that our reweighting heuristics can effectively reduce the number of migrating tasks as tasks are added to or removed from the system.

Organization. The rest of this paper is organized as follows. We describe our system model in Sec. 2, and describe the EDF-sc algorithm in Sec. 3. We derive tardiness bounds for EDF-sc in Sec. 4. These tardiness bounds do not depend on the task assignment being performed in any particular way. Accordingly, numerous heuristics can be used to assign tasks to processors. We present several such heuristics in Sec. 5. We show the results of an experimental evaluation of EDF-sc in Sec. 6. Sec. 7 concludes the paper and outlines future work.

2 SYSTEM MODEL

We consider the scheduling of a dynamic system of sporadic tasks on \( m \) identical processors \( \pi_1, \pi_2, \ldots, \pi_m \) (we assume familiarity with the periodic and sporadic task models). In this paper, we limit our attention to a form of dynamic task system that is commonly found in practice in which tasks may be added to or removed from the system at runtime, but a task’s parameters may not be arbitrarily changed. While some prior work has considered fine-grained reweighting in which task parameters may be arbitrarily changed at runtime [10], our notion of a dynamic task system is sufficient for many real-world use cases such as mode changes. We model such a dynamic task system by a set \( \mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_N\} \) containing all tasks that can ever be run on the system, and a set \( \tau \subseteq \mathcal{T} \) containing the tasks that are currently able to be executed. While \( \mathcal{T} \) is constant, the subset \( \tau \) may change over time.\(^1\)

Each task \( \tau_i \in \mathcal{T} \) is specified by the parameters \( (C_i, T_i) \), where \( C_i \) denotes \( \tau_i \)'s execution cost (assumed to be the worst case) and \( T_i \) denotes its period, or minimum inter-release time. The utilization or weight\(^2\) of task \( \tau_i \) is denoted by \( U_i = C_i / T_i \), and the utilization of a set of tasks \( S \) is denoted \( U(S) = \sum_{\tau \in S} U_i \). We consider dynamic task systems for which \( \forall \tau_i \in \mathcal{T} : U_i \leq 1 \) and \( U(\tau) \leq m \) both hold, as otherwise tardiness may grow without bound. We call an SRT scheduling algorithm optimal if it guarantees that tardiness is bounded for any task system satisfying these conditions.

We denote the \( j \)th job of task \( \tau_i \) by \( \tau_{i,j} \). We denote the release time and deadline of \( \tau_{i,j} \) by \( r_{i,j} \) and \( d_{i,j} \), respectively. We assume all task deadlines are implicit \( (d_{i,j} = r_{i,j} + T_i) \). A job \( \tau_{i,j} \) is called pending at time \( t \) if \( t \geq r_{i,j} \) and \( \tau_{i,j} \) has not completed by \( t \). Assuming that a job \( \tau_{i,j} \) completes execution at time \( t \), its tardiness is \( \max(0, t - d_{i,j}) \). The tardiness of task \( \tau_i \) is the maximum tardiness of any of its jobs.

3 EDF-SC

Our goal in designing EDF-sc was to create a semi-partitioned scheduling algorithm that is optimal in the SRT sense, and that supports dynamic task systems. Because the set of tasks in the system can change over time, unlike most other semi-partitioned scheduling algorithms, EDF-sc does not require any particular offline assignment phase. The initial task assignment can be generated simply by adding one task at a time to an initially empty task system. Performing the initial assignment offline may still be of benefit, however, and we provide some discussion of this issue in Sec. 5.

In fact, unlike every prior semi-partitioned scheduling algorithm known to us, EDF-sc has no dependence on any particular method of assigning tasks to processors: bounded tardiness is guaranteed for all tasks under any arbitrary assignment of tasks to processors. This property holds regardless of even how many tasks are assigned as migrating rather than fixed. In this sense, EDF-sc can be viewed as a generalization of both global and partitioned EDF. If all tasks are assigned to processors and none are migrating, then EDF-sc can behave exactly as partitioned EDF. Similarly, if no tasks are assigned to processors, then EDF-sc behaves exactly as GEDF, maintaining bounded tardiness for all tasks [16].

3.1 Execution

The EDF-sc scheduling algorithm divides the currently active tasks \( \tau \) into \( m + 1 \) pairwise-disjoint subsets \( F_1, F_2, \ldots, F_m, \tau^M \) such that \( \bigcup_{i=1}^m F_i \cup \tau^M = \tau \), and for each \( F_i \), \( U(F_i) \leq 1 \) holds. These sets are depicted in Fig. 1. We call each set \( F_i \) a container. By the scheduling rules of EDF-sc described below, the tasks in each container \( F_i \) can only be scheduled on processor \( \pi_i \). Thus, we say that the tasks in \( F_i \) are fixed on processor \( \pi_i \), while the tasks in \( \tau^M \) are migrating.

We manage the execution budget of each container \( F_i \) using a synchronous and periodic container task \( \tau_{F_i} \) with a period \( T_{F_i} \) and utilization \( U_{F_i} \). From these two parameters, the container task’s

\(^{1}\)Because the set \( \tau \) (and several other sets defined in Secs. 3 and 5) changes over time, it may be more technically precise to use the notation \( \tau(t) \), but we omit the time parameter where it is obvious to avoid clutter.

\(^{2}\)In prior work on dynamic task systems, the term weight is often used to refer to task utilizations. Changing a task’s utilization is referred to as reweighting the task [10, 11].
UWS runs on which their contained tasks are fixed. The scheduling rules at the bottom level of the hierarchy, performing the intra-container containers, and the fixed tasks they contain, from "gaining" tardiness that are fully utilized by their containers. This prevents these to migrate to another processor. Rule S2 fully partitions the processors using GEDF on \(g\) processors. Because \(F_1\) always runs on \(\pi_1\), it may force a job of a migrating task running on \(\pi_1\) to migrate to another processor. Rule S2 fully partitions the processors that are fully utilized by their containers. This prevents these containers, and the fixed tasks they contain, from "gaining" tardiness due to competition from migrating tasks. Rule S3 schedules the bottom level of the hierarchy, performing the intra-container

### Figure 2: Legend for Figs. 3, 4, 5, and 6.

- **Scheduled at top level**: Job release
- **Scheduled at bottom level**: Job completion
- **Execution (\(\pi_i\))**: Execution (\(\pi_i\))
- **Execution (\(\pi_2\))**: Execution (\(\pi_2\))
- **Execution (\(\pi_3\))**: Execution (\(\pi_3\))
- **Execution (\(\pi_4\))**: Execution (\(\pi_4\))

The execution budget is calculated as \(C_{F_i} = U_{F_i} \cdot T_{F_i}\). Unlike the tasks in \(T\), the utilization of each container task may be varied over time in order to adjust the container’s budget. We refer to each time instant \(d_{F_i,j} = r_{F_i,j+1}\) as a job boundary of \(r_{F_i}\). The set of all container tasks is denoted \(r_i\). The period of each container task may be chosen freely and independently by the system designer, so long as \((\forall r_{F_i} \in r_i : T_{F_i} > 0)\) holds. The utilizations for all container tasks must be chosen so that the following two conditions are met.

\[
(\forall r_{F_i} \in r_i : U(F_i) \leq U_{F_i} \leq 1) \\
U(\pi_1) + U(\pi_2) \leq m
\]

If \(U_{F_i} = U(F_i)\), then the container task \(r_{F_i}\) is said to be minimally provisioned; otherwise, it is over-provisioned. If \(U_{F_i} = 1\), then \(r_{F_i}\) is said to be fully provisioned. Note that a container task may be over-provisioned without being fully provisioned, and vice versa.

Let \(r^{m} = \{F_i \mid U_{F_i} = 1\}\) (partitioned container tasks) be the set of fully provisioned container tasks. Note that by construction, \(U(r^{m}) = |r^{m}|\). Also, let \(r^{g} = \{F_i \mid U_{F_i} < 0\}\) (globally scheduled container tasks) be the set of container tasks that are not fully provisioned, and let \(r^{g} = \{\pi_i \mid U_{F_i} < 1\}\) be the processors on which their contained tasks are fixed. The scheduling rules used by EDF-sc are as follows.

**S1** All jobs of tasks in \(r^{g} = r^{m} \cup r^{g}\) are scheduled on a GEDF basis on the processors in \(\pi^{d}\). If a job of a container task \(r_{F_i}\) is selected to run, then it is scheduled on \(\pi_i\).

**S2** Jobs of each container task \(r_{F_i} \in r^{p}\) are scheduled on \(\pi_i\) without competition from migrating tasks.

**S3** When a container task \(r_{F_i}\) is scheduled, it executes the pending job (if any) of a fixed task \(r_{j} \in F_i\) with the earliest deadline. If there is no pending job of any task in \(F_i\), then the container may execute a pending job of a migrating task instead. If no such job is available, then \(\pi_i\) is left idle.

In Rules S1 and S3, all deadline ties are broken in an arbitrary and consistent manner. If two pending jobs \(r_{i,j} \) and \(r_{k,t}\) have the same deadline and \(r_{i,j}\) is prioritized over \(r_{k,t}\) at time \(t\), then \(r_{i,j}\) is prioritized over \(r_{k,t}\) at all other times \(t' \neq t\) when both are pending.

Being container-based, EDF-sc is a hierarchical scheduler with two levels. Rules S1 and S2 prescribe the top level of the scheduling hierarchy, together handling all migrating and container tasks. Rule S1 schedules all migrating tasks and the containers with which they share processors using GEDF on \(\pi^{d}\) processors. Because \(F_1\) always runs on \(\pi_1\), it may force a job of a migrating task running on \(\pi_1\) to migrate to another processor. Rule S2 fully partitions the processors that are fully utilized by their containers. This prevents these containers, and the fixed tasks they contain, from "gaining" tardiness due to competition from migrating tasks. Rule S3 schedules the bottom level of the hierarchy, performing the intra-container scheduling of all fixed tasks. If a container is scheduled but none of its fixed tasks have pending jobs, then its unused budget may be used to execute migrating tasks. This may occur if the container is over-provisioned, or if some of its fixed tasks release jobs with an inter-release time greater than their periods or with an execution cost less than the worst-case value for their tasks.

**Example 3.1.** Fig. 3 shows a four-processor EDF-sc schedule of the tasks \(r_1 = (1, 2)\), \(r_2 = (2, 4)\), \(r_3 = (4, 5)\), \(r_4 = (2, 3)\), \(r_5 = (4, 6)\), and \(r_6 = (2, 3)\), partitioned so that \(F_1 = \{r_1, r_2\}\), \(F_2 = \{r_3\}\), \(F_3 = \{r_4\}\), \(F_4 = \{r_5\}\), and \(r^{m} = \{r_6\}\). All container tasks \(r_{F_i}\) have period \(T_{F_i} = 6\). \(U_{F_1} = U_{F_2} = 1\), and \(U_{F_3} = U_{F_4} = 2/3\). Deadline ties are broken by first prioritizing container tasks over migrating tasks, and then by prioritizing the highest row.

Container tasks \(r_{F_i}\) and \(r_{F_j}\) are fully provisioned, so they are scheduled continuously by Rule S2. The other container tasks compete on a GEDF basis with \(\pi_6\) by Rule S1. \(U(F_3) = 4/5 < U_{F_1}\), so \(\pi_2\) is idle over several intervals. During one of these, \((9, 10)\), jobs of both \(r_{F_1}\) and \(r_{F_2}\) are scheduled by GEDF, but because there are no pending jobs of any task in \(F_2\), \(\pi_6\) is scheduled on \(\pi_2\).

Because the migrating and container tasks in EDF-sc are scheduled at the top level by GEDF, tardiness bounds easily follow from prior work [16]. Bounded tardiness for fixed tasks likewise follows from uniprocessor scheduling analysis with limited processor availability [29]. Thus, for static sporadic task systems, the SRT optimality of EDF-sc (bounded tardiness if no over-utilization) is not hard to show. However, the simplicity of EDF-sc enables the design of rules to support dynamic workload changes, which complicate the derivation of tardiness bounds somewhat. We prove expressions for such bounds in Sec. 4, after first describing in the following section the rules we propose for managing dynamic workload changes.
3.2 Reweighting Rules

EDF-sc supports dynamic task systems by allowing tasks to be added to or removed from the system. This is done by applying the following reweighting rules, which support the addition and removal of tasks in $\tau$, as well as changing container weights. The rules described in this section are low-level operations, and in some cases more than one rule may need to be applied to add a given task to the system. In Sec. 5, we give heuristics that handle these cases seamlessly and attempt to assign all tasks as fixed where possible.

**AM (add migrating task)** A task $\tau_a \not\in \tau$ may be added to $\tau^M$ if and only if the following condition holds.

$$U_a \leq \min\{1, |\pi^a| - U(\pi^a)\}$$

**AF (add fixed task)** A task $\tau_a \not\in \tau$ may be added to the container $F_i$ if and only if the following condition holds.

$$U_a \leq U_{F_i} - U(F_i)$$

**R (remove task)** A task $\tau_i \in \tau$ whose most recently released job $\tau_{r,i}$ completes at time $t_c$ may be removed from $\tau$ at or after time $\max(d_{r,i}, t_c)$, provided no new job $\tau_{r,i+1}$ is released before this time.

**W (reweight container)** The weight of a container task $\tau_{F_i}$ may be changed from $U_{F_i}$ to $U'_{F_i}$ at its job boundaries if and only if the following two conditions hold.

$$U(F_i) \leq U'_{F_i} \leq 1$$
$$U'_{F_i} - U_{F_i} + U(\tau^M \cup \tau^F) \leq m$$

The weight change only affects newly released jobs of $\tau_{F_i}$.

Intuitively, Rule AM allows a migrating task to be added to the task system if it does not over-utilize a single processor, and if its addition will not cause the processors of $\pi^a$ to be over-utilized by containers and migrating tasks. Similarly, Rule AF allows a fixed task to be added to a container if adding it will not cause the container to be over-utilized. Rule R allows tasks to be removed at job deadlines or completion times, whichever is later. Rule W allows container tasks to be reweighted at container job boundaries as long as the new utilization is at least the utilization of the contained tasks, and the system is not over-utilized at the top level.

4 TARDINESS BOUNDS

In this section, we derive tardiness bounds for all tasks in a dynamic sporadic task system scheduled by EDF-sc. We begin in Sec. 4.1 by proving a tardiness bound for migrating and container tasks based on work by Devi [15]. We then use this bound to derive a tardiness bound for fixed tasks under EDF-sc in Sec. 4.2.

4.1 Tardiness Bound for Migrating and Container Tasks

By Rule S3, the migrating and container tasks in EDF-sc compete on a GEDF basis on $|\pi^a|$ processors. Rule S3 may cause migrating tasks to be scheduled by the container tasks, but this can only move execution of migrating tasks earlier, so Rule S3 can never increase the tardiness of migrating or container tasks. The set of globally scheduled tasks can be modified by Rules AM, R, and W.

Devi [15] presented an extended sporadic task model that can be used to model dynamic task models such as the one considered in this work. In Devi’s model, the total utilization of all tasks is allowed to exceed $m$, but at any time instant, the total utilization of all active tasks is at most $m$. A task is initially inactive until the release of its first job, at which point it is active until the deadline of its last job, when it becomes terminated. The extended sporadic task model divides the tasks in the system into task classes $\tau^1, \tau^2, \ldots, \tau^n$ so that the active intervals for each pair of tasks in each class are disjoint, and with the precedence constraint that the first job of a task cannot execute until all jobs of tasks in the same class with earlier release times have completed. Intuitively, each task class in this model is meant to model a single dynamic task, and each task in a class represents a set of parameters it could take on. Devi showed that using GEDF scheduling in this extended sporadic task model, tardiness for each task in the task class $\tau^i$ is at most

$$\sum_{k=1}^{m} \max\{0, (k-1)C^\text{max}_{k-1} - C^\text{max}_{k}\},$$

where $C^\text{max}_{k}$ are the maximum execution cost and utilization of any task in task class $\tau^i$ and $\ell^\text{max}_{k}$ are the subsets of $\ell$ task classes with the greatest values of $C^\text{max}$ and $U^\text{max}$, respectively.

**Example 4.1.** An example of an extended sporadic task system scheduled on two processors by GEDF is depicted in Fig. 4. This task system consists of three task classes $\tau^1 = (\tau_1 : (2, 4)), \tau^2 = (\tau_2 : (2, 3)), \text{ and } \tau^3 = (\tau_3 : (5, 5), \tau_4 : (4, 5))$. Before time 10, only tasks $\tau_1$ and $\tau_3$ are active, giving a system utilization of 3/2. At time 10, $\tau_3$ terminates and $\tau_2$ and $\tau_4$ activate, increasing the system utilization to 59/30. At time 16, task $\tau_2$ terminates, decreasing the system utilization to 13/10.

**Lemma 4.2.** Migrating and container tasks in EDF-sc can be modeled by Devi’s extended sporadic task model.
Theorem 4.3. Tardiness for any migrating or container task \( \tau_i \) under EDF-sc is at most

\[
\frac{\sum_{\tau_k \in \pi_{\text{max}}(\tau^m \cup \tau^f)} |\tau_k| C_{\text{max}}}{m - \sum_{\tau_k \in \pi_{\text{max}}(\tau^m \cup \tau^f), m-2} U_{\text{max}}^i} + c_{\text{max}}^i, \tag{8}
\]

where \( c_{\text{max}}^i \) and \( U_{\text{max}}^i \) are the maximum execution cost and utilization of task \( \tau_i \), and \( \pi_{\text{max}}(\tau, \ell) \) and \( \pi_{\text{max}}^i(\tau, \ell) \) are the subsets of \( \ell \) tasks in \( \tau \) with the greatest values of \( C_{\text{max}}^i \) and \( U_{\text{max}}^i \), respectively.

Proof. Rule S1 schedules migrating and container tasks using GEDF, so as long as this is the only rule that schedules migrating and container tasks, the bound (8) follows from Lemma 4.2.

Rule S3 can schedule migrating tasks using the budget of container tasks. This does not directly affect the tardiness of container tasks, as they consume budget regardless. The tardiness of migrating tasks can only be decreased by Rule S3, because their execution can only be moved earlier by this rule. Thus, the bound (8) holds if migrating and container tasks are scheduled by Rules S1 and S3.

It remains to be shown that the bound (8) holds when container tasks are scheduled by Rule S2 as well. This rule schedules fully provisioned container tasks without competition from migrating tasks, so their tardiness can never increase as a result of being scheduled by Rule S2. This causes the tasks in \( \tau^m \) to be scheduled using GEDF on \( |\pi^m| < m \) processors. Thus, by Lemma 4.2, tardiness for a task \( \tau_i \in \tau^m \) is upper-bounded by

\[
\frac{\sum_{\tau_k \in \pi_{\text{max}}(\tau^m \cup \tau^f)} |\tau_k| C_{\text{max}}}{|\pi^m| - \sum_{\tau_k \in \pi_{\text{max}}(\tau^m \cup \tau^f), |\tau_k| < 2} U_{\text{max}}^i} + c_{\text{max}}^i. \tag{9}
\]

The numerator of (9) is at most the numerator of (8) because it is the sum of fewer maximum execution costs from a subset of the set considered in (8). The denominators of the two expressions are equal: the tasks in \( \tau^m \cup \tau^f \) excluded from the summation in the denominator of (9) are the fully provisioned containers \( \tau^p \), whose total maximum utilization is \( m - |\tau^f| \). Therefore, tardiness for tasks in \( \tau^m \) is still upper-bounded by (8), so the theorem holds.

4.2 Tardiness Bound for Fixed Tasks

Fixed tasks are scheduled using uniprocessor EDF by Rule S3, but they may still miss deadlines if their container is not fully provisioned. This is because the processor is unavailable to the fixed tasks when their container task is not scheduled, which could cause processor demand to exceed supply over some time intervals. Real-time scheduling with limited processor availability has been studied previously [28, 29], but prior work focuses on static systems. In this section, we derive a tardiness bound for fixed tasks under EDF-sc by extending limited availability analysis techniques to handle dynamic behaviors. To aid in this, for the remainder of this section, we will explicitly show the time parameter \( t \) for container \( F_i(t) \) and the utilization of its container task, \( U_{F_i}(t) \).

Theorem 4.4. Under EDF-sc, tardiness for fixed tasks on processor \( \pi_i \) is at most

\[
\sigma_i = 2T_{F_i} + \frac{\sum_{\tau_k \in \pi_{\text{max}}(\tau^m \cup \tau^f), m-1} |\tau_k| C_{\text{max}}}{m - \sum_{\tau_k \in \pi_{\text{max}}(\tau^m \cup \tau^f), m-2} U_{\text{max}}^i} + c_{\text{max}}^i. \tag{10}
\]

Proof sketch. Assume for purposes of contradiction that in a system scheduled by EDF-sc, some job \( r_{a,b} \) of a fixed task \( r_a \) on processor \( \pi_i \) completes with tardiness greater than \( \sigma_i \). In particular, let \( t_c > d_{a,b} + \sigma_i \) denote the completion time of \( r_{a,b} \). Define a job \( r_{i,j} \) of a fixed task on processor \( \pi_i \) as a competing job if its deadline is at or before \( d_{i,j} \). Let \( t_0 \) denote the unique instant in time at or before \( r_{i,j} \) immediately before which no competing jobs were pending, and starting at which \( \pi_i \) is either busy executing competing jobs or unavailable to fixed tasks until \( t_c \).

We next derive bounds for processor demand and supply over any time interval \( [r, s) \), where for every \( t \in [r, s) \), \( U(F_i(t)) > 0 \) holds. Because the interval of interest \( [t_0, t_c] \) meets this description, we can then use these bounds to derive a completion time for \( r_{a,b} \). As discussed earlier, due to the dynamic behavior of EDF-sc, we cannot simply use linear bounds for these functions. Instead, we derive piecewise linear functions expressed with definite integrals.
The processor demand created by jobs of fixed tasks on \( \pi_i \) over an interval \([r, s]\) is at most
\[
a_i(r, s) = \int_r^s U(F_i(t)) \, dt.
\]
Therefore, the processor demand created by competing jobs released at or after \( t_0 \) is at most \( a_i(t_0, d_i) \).

Now we derive a lower bound to processor availability to fixed tasks on \( \pi_i \) over an interval \([r, s]\), as described above. Prior work involving limited processor availability for static systems typically provides a lower bound for an interval of length \( \Delta \) of the form
\[
\beta(\Delta) = \max\left(0, \frac{U - \sigma}{\Delta} \cdot (\Lambda - \sigma) \right),
\]
where \( \bar{U} \) is the long-term average processor availability, and \( \sigma \) is the maximum duration of time where the processor can be continuously unavailable [28]. This is insufficient for our needs because the average processor availability can change over time, so we develop a similar lower bound to availability over an interval as described above.

By examining the scheduling of \( T_i \), we can see that the maximum processor unavailability occurs when a job \( T_{i,j} \) completes execution as early as possible, and the next job \( T_{i,j+1} \) completes as late as possible. Thus by adding two container periods to the tardiness bound from Theorem 4.3, we arrive at \( \sigma_i \) from the theorem statement as an upper bound to the maximum duration of processor unavailability regardless the container task’s utilization.

We now argue that the processor availability over any interval \([r, s]\) is at least
\[
\beta_i(r, s) = \max\left(0, \int_r^s U(F_i(t)) \, dt \right).
\]

The reweighting rules in Sec. 3.2 guarantee that at any time \( t \), \( U(F_i(t)) \leq U_i(t) \), so it is safe to use the utilization of the contained tasks \( U(F_i(t)) \) in a lower bound for processor availability. The function \( \beta_i(r, s) \) assumes that the processor is initially unavailable for \( \sigma_i \) time units. Because this occurs when a job \( T_{i,j} \) completes as late as possible, the entire budget of \( T_{i,j} \) must then be consumed non-preemptive. Furthermore, to ensure the tardiness bound is met, each subsequent job \( T_{i,j+k} \) where \( k > 0 \) must now consume its budget over an interval of length at most \( T_i \), maintaining an average availability of \( U_i(t) \) from that moment onward.

Now that we have derived an upper bound to processor demand and a lower bound to processor supply, we observe that the maximum demand created by competing jobs released at or after \( t_0, \sigma_i(t_0, d_{a,b}) \), equals the minimum processor supply from \( t_0 \) to \( d_{a,b} + \sigma_i \). Therefore, all work created by competing jobs must have completed by time \( d_{a,b} + \sigma_i \). This contradicts the assumption that \( T_{a,b} \) completes at time \( t_e \geq d_{a,b} + \sigma_i \), so the theorem holds.

## 5 REWEIGHTING HEURISTICS

The tardiness bounds shown in Sec. 4 apply for any assignment of tasks to processors, as long as no processor or the whole system is ever over-utilized. Therefore, the rules in Sec. 3.2 can be composed to create reweighting heuristics that are stabilizing; that is, as tasks are added to and removed from the system, the heuristics attempt to fully partition the workload.

### 5.1 Initial Assignment

If the initial set of tasks \( r \) is known in advance, a static assignment of tasks to processors to be used at system startup may be produced offline. To create this initial assignment, we suggest that the system designer try several different bin-packing heuristics and compare their results. Clearly any produced assignment that makes all tasks fixed is preferable to one that does not, as such an assignment enables EDF-sc to operate as partitioned EDF, guaranteeing zero tardiness for all tasks until the first reweighting event. If multiple assignments are able to fix all tasks to processors, then it is likely that an assignment whose least-utilized processor has the lowest utilization among all assignments is preferable, as this would allow tasks with higher utilization to be added to the system later without having to migrate. If no assignment is able to fix all tasks to processors, then one that allows the most containers to be fully provisioned would tend to give lower tardiness bounds not only for the fixed tasks in these containers, but for all other tasks in the system as well. This is because GEDF tardiness bounds tend to be lower (and tighter) with smaller processor counts [15].

### 5.2 Runtime Workload Changes

When making workload changes at runtime, a system designer may only be concerned with the high-level operations of adding and removing tasks. By contrast, the EDF-sc reweighting rules in Sec. 3.2 expose the details of whether a new task is to be fixed or migrating, which container a fixed task is added to, and updating container utilizations. In this section, we propose a set of reweighting heuristics to support high-level “add task” and “remove task” operations. These heuristics try to assign as many tasks as possible as fixed, and to keep as many containers fully provisioned as possible. To ease the process of reweighting containers due to dynamic workload changes, the heuristics proposed here require that all container tasks have equal periods.

Additionally, when fixed tasks are removed, the heuristics attempt to move migrating tasks that are already in \( r \) into containers. We refer to this process as stabilizing the workload. The problem of stabilizing a dynamic semi-partitioned workload is related to the problem of fully dynamic bin-packing [23]. However, approximation algorithms for fully dynamic bin packing are designed to operate on an infinite number of bins, and cannot be directly applied to our problem where the number of bins is finite and the goal is to pack as many items as possible rather than to pack all items into the fewest possible bins. Pseudocode for our reweighting heuristics is listed in Algorithm 1, which we explain next.

**Task addition.** Requests to add tasks to \( r \) may be issued at any time using the procedure \( \text{AddTask}(T_k) \). Task additions are not actually carried out by our heuristic until container job boundaries because containers may first need to be reweighted; this is discussed in more detail below. The \( \text{AddTask} \) procedure simply adds the task \( T_k \) to a FIFO queue that holds all tasks that have been requested to be added since the last container period boundary.

At each container job boundary, the queue is emptied in the while loop at lines 11–14, running a \( \text{ContainerSelectionHeuristic} \) for each task in FIFO order to attempt to add tasks into temporary containers \( F_i' \). This heuristic uses a bin-packing heuristic to find a container \( F'_i \) so that \( U(F'_i) + U_a \leq 1 \) holds.

5 REWEIGHTING HEURISTICS

The tardiness bounds shown in Sec. 4 apply for any assignment of tasks to processors, as long as no processor or the whole system is ever over-utilized. Therefore, the rules in Sec. 3.2 can be composed to create reweighting heuristics that are stabilizing; that is, as tasks are added to and removed from the system, the heuristics attempt to fully partition the workload.
Algorithm 1 Heuristics for adding and removing tasks, and for stabilizing the workload.

PendingAdds: FIFO queue, initially empty

1: procedure AddTask(τ_u)
2: PendingAdds.Enq(τ_u)

3: procedure RemoveTask(τ_r)
4: if τ_r’s most recently released job τ_{r,j} is pending then
5: Forget τ_r from releasing new jobs
6: Schedule R(τ_r) to occur at the later of d_{r,j} and τ_{r,j}’s completion time
7: else
8: R(τ_r) ← Remove the task now

9: procedure ContainerBoundary()
10: Function called at each container job boundary t
11: F_{t}, \ldots, F_{m}, t^* \leftarrow F_{t}, \ldots, F_{m}, t^*
12: Determine which tasks can be added
13: while PendingAdds is not empty do
14: τ_u \leftarrow PendingAdds.Deq()
15: Select container F'_{u}, t^*, or ⊥ for τ_u via ContainerSelectionHeuristic
16: Add τ_u to selected container, or reject if ⊥
17: Try to move migrating tasks into containers
18: for each τ_k \in τ^d do
19: if τ_k’s most recently released job τ_{k,j} is not pending and d_{k,j} < t + T_{F_1} then
20: and τ_{k,j} will fit in some container F'_k then
21: Select container F'_{k} via ContainerSelectionHeuristic
22: Add τ'_{k,j} to selected container
23: Schedule an atomic operation (R(τ'_{k,j}; R(τ_k); AF(τ_{k,j}, F_1))) to occur at max(d_{k,j}, t), i.e., use Rules R and AF to enact the move of τ_{k,j} into F_1
24: Compute new container task weights via ContainerReweightHeuristic
25: Reweight container tasks using Rule W
26: Add τ_{k,j} to that container at line 19. We then schedule an atomic operation at line 20 to remove τ'_{k,j} from the container, remove τ_k as a migrating task, and add τ_{k,j} to the container at the deadline of its most recently released job.

In our experiments in Sec. 6, we evaluate the choices of BestFit, WorstFit, and FirstFit as the bin-packing heuristic. If no suitable container can be found, ContainerSelectionHeuristic checks if τ_u can be added as a migrating task by checking if \sum_{i=1}^{m} U(F_i) + U(t^*) + U_t \leq m holds. If not, the heuristic returns ⊥, and τ_u is rejected.

After the new tasks are added to the temporary containers, the container tasks are reweighted with Rule W, which we discuss below. Following this, all new tasks are added at lines 23–27 using the EDF-sc reweighting rules for adding tasks.

Task removal. Requests to remove tasks from τ may be issued at any time using the procedure RemoveTask. This procedure simply removes task τ_r with Rule R as early as possible, preventing it from releasing new jobs if necessary.

Stabilization. At each container job boundary, after accepting any new tasks to be added to the system, the heuristics attempt to stabilize the workload by moving migrating tasks into containers, making them fixed tasks (lines 15–20). The procedure by which we move tasks is illustrated in Fig. 6. Using the rules from Sec. 3.2, this must be done by removing a migrating task τ_u using Rule R and immediately adding it as a fixed task using Rule AF. This could potentially prevent job releases of τ_k if its last released job τ_{k,j} completes at a future time after its deadline, so to avoid this, we require that τ_{k,j} is not pending when deciding to move τ_k. (While we deem such delayed job releases as unacceptable, applications might exist in which they can be tolerated, in which case alternative heuristics could be used.)

If we decide to move task τ_k into a container, then we can only do so at the later of its deadline or the current time. Therefore, we must ensure that there is space in the container to add τ_k at its deadline. This is accomplished creating a copy of τ_k at line 16, denoted τ'_{k,j}, which never releases any jobs and merely serves as a placeholder for τ_k. Because this space must be reserved over the time interval between deciding to move τ_k and enacting the move, we do not attempt to move any tasks whose last-released job has a deadline more than one container period in the future, as this would keep the container’s utilization reserved for longer than necessary. (Alternative heuristics could resolve this issue differently.)

For each task τ_k that is eligible to be moved by the conditions at line 17 outlined above, we choose a container for it using the ContainerSelectionHeuristic at line 18, then add its copy τ'_{k,j} to that container at line 19. We then schedule an atomic operation at line 20 to remove τ'_{k,j} from the container, remove τ_k as a migrating task, and add τ_{k,j} to the container at the deadline of its most recently released job.

Container task reweighting. Once all operations that add tasks to containers have been fully planned, we call a ContainerReweightHeuristic at line 21 to determine new container weights based on the sets F'_{1}, \ldots, F'_{m}, t^*'. This procedure determines new weights for the container tasks so that for each τ_F, U(F_i) \leq U_F \leq 1 holds, and so that \sum_{i=1}^{m} U_F + U(t^*) \leq m holds. There are many different ways this reweighting can be done; we propose two techniques here and evaluate them experimentally in Sec. 6.2.

**MINOrFULL** (Fig. 7a) Begin by minimally provisioning all container tasks. Then fully provision as many as possible while maintaining U(t^*) \leq |τ^d|, e.g. by choosing container tasks in order of decreasing U(F_1). This may leave some processor capacity in τ^d unavailable to fixed tasks.

**EQUALOver** (Fig. 7b) Fully provision container tasks as with MINOrFULL, but over-provision the others, each by an equal fraction of the extra processor capacity |τ^d| – U(t^*).

Once the new weights have been determined by one of these heuristics, the container tasks are all reweighted using Rule W at line 22.
Figure 7: The two container task provisioning techniques discussed. (a) Containers \( \tau_{F_i} \) and \( \tau_{F_4} \) are minimally provisioned \( (U_{F_i} = U(F_i)) \), and \( \tau_{F_i} \) and \( \tau_{F_4} \) are fully provisioned \( (U_{F_i} = 1) \). (b) Container task utilizations \( U_{F_i} \) and \( U_{F_4} \) are increased equally to consume the system capacity left unused by MinOrFull.

6 EXPERIMENTS

In this section, we present the results of experiments conducted to evaluate EDF-sc, compared to GEDF, which is also SRT-optimal and supports dynamic task systems. We first demonstrate the advantages of semi-partitioned scheduling over global scheduling with a cache-related preemption and migration delay (CPMD)-aware schedulability study in static systems. We then evaluate the effectiveness of the various options presented in Sec. 5 for bin-packing heuristics and container task provisioning in dynamic systems. We also compare observed tardiness and tardiness bounds for dynamic systems under EDF-sc and GEDF. Finally, we demonstrate how effectively our heuristics stabilize a dynamic workload by comparing versions of Algorithm 1 with lines 15–20, which attempt to move migrating tasks into containers, either enabled (i.e., heuristics are applied) or disabled (i.e., no heuristics applied). Code for all the experiments in this section is available online [22].

6.1 CPMD-Aware Schedulability Study

To illustrate the advantages of semi-partitioned scheduling over global scheduling, we performed a CPMD-aware schedulability study comparing EDF-sc to GEDF with a variety of static task systems. In lieu of a kernel implementation of EDF-sc (the development of which we leave to future work), we cannot consider overheads due to the execution of the scheduler itself. However, since semi-partitioning results in run queues that are smaller than a single, global run queue, we expect overheads in a static system to be lower in EDF-sc than in GEDF. The reweighting heuristics may offset this advantage somewhat, so we expect the overheads to be similar between EDF-sc and GEDF in a dynamic system.

In these experiments, we randomly generated static sporadic task systems and increased the execution times of each task to account for average-case CPMD overheads on a 24-core Intel system as measured in [5]. We then calculated the schedulability, or proportion of the generated task systems that remain feasible, under each scheduler. Task sets were generated similarly to [6], with task utilizations chosen from uniform, bimodal, and exponential distributions, each with medium or heavy tasks. For uniform distributions, task utilizations were chosen from [0.1, 0.4] for medium task sets and from [0.5, 0.9] for heavy task sets. For bimodal distributions, task utilizations were chosen uniformly from [0.001, 0.5) or [0.5, 0.9] with probabilities of 6/9 and 3/9 for medium task sets and 4/9 and 5/9 for heavy task sets, respectively. For exponential distributions, task utilizations were generated with a mean of 0.25 for medium task sets and 0.5 for heavy task sets, discarding any values greater than 1. For each of these, we generated task periods uniformly at random from [3, 33] ms (short), [10, 100] ms (moderate), or [50, 250] ms (long). We generated task sets with utilization caps in [15, 24] with a step size of 0.25, and with working set sizes (WSS) of each power of two in [64, 2048] KB.

For each combination of utilization distribution, period distribution, utilization cap, and WSS, we generated 100 task sets by adding tasks until the next task would cause the utilization cap to be exceeded without accounting for overheads. The schedulers tested were EDF-sc with BestFit bin-packing and minimal container provisioning (we do not expect the choice of heuristics to have a major impact on schedulability), and GEDF. For EDF-sc, we used a binary search over [0, 1024] ms to determine the minimum container task period for which each task set was schedulable, and computed the average of these for each task set parameter combination. We considered CPMD values for both idle systems (low cache contention) and systems under load (high cache contention). To aid in the presentation of a large number of schedulability experiments, we use weighted schedulability [5], defined as 

\[
S(W) = \frac{\sum_{U \in Q} U \cdot S(U, W))}{\sum_{U \in Q} U}, \quad \text{where } Q \text{ is the set of utilization caps considered, } W \text{ is a WSS value, and } S(U, W) \text{ is the schedulability ratio for utilization cap } U \text{ and WSS } W.
\]

Results. Due to space constraints, the full results of these experiments are given in an online appendix [22]. A typical result for task sets with short periods is shown in Fig. 8. Systems with moderate and long periods show a similar trend, but weighted schedulability is improved about uniformly for all schedulers considered. In general, GEDF gave significantly lower schedulability than EDF-sc with idle overheads, and slightly lower schedulability with load overheads. Due to several effects influencing the measurement of CPMD values, GEDF gave higher schedulability with larger WSS values when using load overheads instead of idle overheads; please see [5] for more details. Further, the container task periods remained very low (typically under about 30 ms) for most of the feasible task sets, increasing sharply for higher utilizations where most task sets were unschedulable. The plots for this are given in an online appendix.
6.2 EDF-sc Heuristic Comparison

To evaluate the efficacy of our reweighting heuristics for EDF-sc, we conducted experiments on a simulated identical 24-processor system running a set of synthetically generated dynamic workloads. Intuitively, one heuristic is more effective than another if it provides reduced tardiness in a dynamic workload. To quantify this, we measured the average and maximum observed tardiness of all jobs in each dynamic workload, and the analytic tardiness bounds that result. We conducted the same experiments with GEDF, using EDF-sc rules AM and R to add and remove tasks from the system. Using GEDF this way is actually a special case of EDF-sc where all container tasks always have utilization 0.

Dynamic workload generation. For this experiment, we generated dynamic workloads using a similar methodology to one pioneered by Casini et al. [14]. Each dynamic workload consisted of a task set \( T \), and a sequence of task addition and removal events. The tasks in \( T \) were generated with periods selected from a uniform distribution over the range \([10, 1000]\) ms, and with utilizations generated from a beta distribution with mean \( \mu_U \) and variance \( \sigma_U^2 \). We used values for \( \mu_U \) of 0.2, 0.4, and 0.6 to create workloads that are easy-, moderate-, and hard-to-partition, respectively. Preliminary results showed that changing the value of \( \sigma_U^2 \) did not have a major impact on the results, so we kept this at a constant value of 0.006. In all of our experiments, all tasks released jobs periodically.

Initially the set of active tasks \( r \) in each dynamic workload consisted of tasks from \( T \) chosen at random one at a time until the next chosen task would have become unschedulable by the CPMD-aware schedulability test used in Sec. 6.1. This task set was modified by a sequence of 100 task addition or removal events, with interarrival times selected uniformly over the range \([1000, 4000]\) ms. To determine whether each event should constitute a task addition or removal, we generated a number \( x \) from a uniform distribution over \([0, 1]\), and compared it to a threshold \( \Lambda = (1-U(t)/m) + \psi(U(t)/m) \) that was used to control the average load on the system. If \( x \leq \Lambda \), then the event attempted to add a random task from \( T \setminus r \) to \( r \). Otherwise, the event selected a random task to be removed from \( r \). Generating events in this way increases the probability of adding tasks when the system utilization is low, and increases the probability of removing tasks when the system utilization is high [14].

We held \( \psi = 0.8 \) across all generated workloads to keep system utilization high.

For each combination of \( \mu_U \) value and WSS for each power of two in \([64, 2048]\) kB, we generated 100 dynamic workloads. We simulated the scheduling of each workload on a 24-processor system using EDF-sc, with each combination of the bin-packing heuristics BestFit, WorstFit, and FirstFit, and the container reweighting heuristics MinOrFull and EqualOver. For these experiments, we only used idle CPMD overheads (low cache contention), as this is a more common case in real systems. Following the results of the experiments in Sec. 6.1, we set all container periods to 30 ms in these experiments. We also simulated the scheduling with GEDF to compare tardiness between the two schedulers.

Results. Fig. 9 shows average tardiness across all sets of generated workloads that are easy-, moderate-, and hard-to-partition. In each graph, the x axis shows each WSS considered, and the y axis shows the average tardiness observed across all generated workloads. A lower curve represents a better-performing heuristic. To reduce visual clutter, standard deviations are omitted from the figure. The relative orders of the heuristics by maximum observed tardiness were similar to those for average tardiness, but with values around 300, 100, and 20 times larger for easy-, moderate-, and hard-to-partition workloads, respectively. Due to space constraints, these graphs are given in an online appendix [22], along with the tardiness bounds. GEDF typically gave maximum tardiness around a quarter to a half that of EDF-sc, though for easy-to-partition task sets with higher WSS values, the two algorithms were roughly equal.

We found that across a majority of scenarios considered, the container reweighting procedure MinOrFull gave slightly greater average task tardiness than EqualOver. The BestFit and FirstFit bin-packing heuristics gave similar average tardiness to one another, and WorstFit gave somewhat higher tardiness in most cases. Given the overall trends and considering runtime overhead, we consider the best overall combination of heuristics to be FirstFit bin-packing with EqualOver container task provisioning.

We found that GEDF gave similar average tardiness values regardless the average task utilization. GEDF gave lower average tardiness than EDF-sc for all WSS values in hard-to-partition workloads. However, EDF-sc gave lower average tardiness for higher
Figure 10: Stabilization heatmaps for WSS of 2048 kB. Insets (a), (b), and (c) (top) show easy-, moderate-, and hard-to-partition workloads with stabilization disabled. Insets (d), (e), and (f) (bottom) show easy-, moderate-, and hard-to-partition workloads with stabilization enabled.

WSS values in the moderate- and easy-to-partition workloads, so EDF-sc may be a good choice for tasks with lower utilizations.

6.3 Stabilization
To evaluate how effectively our heuristics stabilize dynamic workloads, we measured the number of migrating tasks at intervals of 1000 ms in 50 workloads generated as in Sec. 6.2, using FirstFit bin-packing and EQualOver container task provisioning. To obtain a baseline, we conducted the same experiments with lines 15–20 of our heuristics, which attempt to move migrating tasks into containers, disabled.

**Results.** Results of these experiments for a WSS of 2048 kB are shown in Fig. 10. Due to space constraints, the other WSSs are included in an online appendix [22]. Insets (a), (b), and (c) show the number of migrating tasks over time with stabilization disabled in easy-, moderate-, and hard-to-partition workloads, respectively, and insets (d), (e), and (f) show the number of migrating tasks over time with stabilization enabled in easy-, moderate-, and hard-to-partition workloads, respectively. Each inset shows the number of migrating tasks over time as a heatmap: the $x$ axis shows time, and the $y$ axis shows the number of migrating tasks. Through trial and error, we found that the average number of migrating tasks had stopped changing before 200 s in all cases, so we limited the $x$ axis to show the interval $[0, 200]$ s. The color of each cell indicates how many workloads had a particular number of migrating tasks at each time observed.

We found that hard-to-partition workloads, the stabilization code had little effect on the number of migrating tasks. Indeed, for task sets with a mean task utilization of 0.6, it would not be possible in many cases to pack two tasks into a single container. However, we also found that for moderate- and easy-to-partition task sets, our stabilization heuristic successfully reduced the number of migrating tasks below the baseline with stabilization disabled.

7 CONCLUSION
We have presented EDF-sc, the first semi-partitioned scheduling algorithm that is optimal for static SRT sporadic task systems (in the “bounded tardiness” sense of SRT correctness) and that can also handle dynamic workload changes. EDF-sc guarantees bounded tardiness for all tasks regardless of the assignment of tasks to processors. If no tasks are assigned as fixed, then it behaves as GEDF, and if all tasks are fixed, then it behaves as partitioned EDF. Because it is desirable to fully partition the task system if possible, we presented heuristics to achieve this goal as tasks are reweighted. These heuristics afford EDF-sc with a novel property, never before considered in work on semi-partitioned scheduling, of being able to stabilize towards increasing the number of tasks that are fixed as the system executes.

Because fixed tasks in EDF-sc execute within containers that may lose budget, they effectively have priorities that may vary over time. Since most multiprocessor real-time locking protocols require job-level fixed priorities, it is not currently clear how to handle shared resources accessible by fixed tasks under EDF-sc. We would like to investigate a job-level fixed priority variant of EDF-sc to ease this restriction.
REFERENCES


A FIGURES

A.1 Schedulability Experiments

Figure 11: Schedulability results for medium task sets generated with a uniform distribution.

Figure 12: Schedulability results for medium task sets generated with a bimodal distribution.

Figure 13: Schedulability results for medium task sets generated with an exponential distribution.
Figure 14: Schedulability results for heavy task sets generated with a uniform distribution.

Figure 15: Schedulability results for heavy task sets generated with a bimodal distribution.

Figure 16: Schedulability results for heavy task sets generated with an exponential distribution.
A.2 Container Period Experiments

Figure 17: Minimum container periods required to schedule medium task sets generated with a uniform distribution and medium periods at different WSS.

Figure 18: Minimum container periods required to schedule medium task sets generated with a uniform distribution and differing periods at WSS = 512.

Figure 19: Minimum container periods required to schedule medium task sets generated with different distributions and medium periods at WSS = 512.
Figure 20: Minimum container periods required to schedule heavy task sets generated with a uniform distribution and medium periods at different WSS.

Figure 21: Minimum container periods required to schedule heavy task sets generated with a uniform distribution and differing periods at WSS = 512.

Figure 22: Minimum container periods required to schedule heavy task sets generated with different distributions and medium periods at WSS = 512.
A.3 Tardiness Experiments

Figure 23: Left to right: observed maximum tardiness over easy-, moderate-, and hard-to-partition workloads.

Figure 24: Left to right: tardiness bound over easy-, moderate-, and hard-to-partition workloads.
A.4 Migrating Task Experiments

Figure 25: Stabilization heatmaps for WSS of 64 kB. Insets (a), (b), and (c) (top) show easy-, moderate-, and hard-to-partition workloads with stabilization disabled. Insets (d), (e), and (f) (bottom) show easy-, moderate-, and hard-to-partition workloads with stabilization enabled.

Figure 26: Stabilization heatmaps for WSS of 128 kB. Insets (a), (b), and (c) (top) show easy-, moderate-, and hard-to-partition workloads with stabilization disabled. Insets (d), (e), and (f) (bottom) show easy-, moderate-, and hard-to-partition workloads with stabilization enabled.
Figure 27: Stabilization heatmaps for WSS of 256 kB. Insets (a), (b), and (c) (top) show easy-, moderate-, and hard-to-partition workloads with stabilization disabled. Insets (d), (e), and (f) (bottom) show easy-, moderate-, and hard-to-partition workloads with stabilization enabled.

Figure 28: Stabilization heatmaps for WSS of 512 kB. Insets (a), (b), and (c) (top) show easy-, moderate-, and hard-to-partition workloads with stabilization disabled. Insets (d), (e), and (f) (bottom) show easy-, moderate-, and hard-to-partition workloads with stabilization enabled.
Figure 29: Stabilization heatmaps for WSS of 1024 kB. Insets (a), (b), and (c) (top) show easy-, moderate-, and hard-to-partition workloads with stabilization disabled. Insets (d), (e), and (f) (bottom) show easy-, moderate-, and hard-to-partition workloads with stabilization enabled.