Statically Optimal Dynamic Soft Real-Time Semi-Partitioned Scheduling

Clara Hobbs · Zelin Tong · Joshua Bakita · James H. Anderson

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.

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

Work supported by NSF grants CNS 1409175, CNS 1563845, CNS 1717589, and CPS 1837337, ARO grant W911NF-17-1-0294, and funding from General Motors.

Clara Hobbs · Zelin Tong · Joshua Bakita · James H. Anderson
Department of Computer Science, UNC-Chapel Hill, 201 S. Columbia St., Chapel Hill, NC 27599
E-mail: {cghobbs,ztong,jbakita,anderson}@cs.unc.edu

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 (Anderson et al. 2005), 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.

1.1 Prior work

The first semi-partitioned scheduling algorithm to be proposed was EDF-fm (Anderson et al. 2005), 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 (Anderson et al. 2016; Andersson et al. 2008; Andersson and Tovar 2006; Bhatti et al. 2012; Bletsas and Andersson 2009, 2011; Brandenburg and Gül 2016; Burns et al. 2012; Casini et al. 2017; Dorin et al. 2010; Fan and Quan 2012; Guan et al. 2010a,b; Kato and Yamasaki 2009, 2008; Shekhar et al. 2012; Kato and Yamasaki 2007; Sousa et al. 2013; Voronov and Anderson 2018). Of these, two are especially relevant to this work. Anderson et al. (2016) 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. (2017) 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.

1.2 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 an overhead-aware schedulability study comparing EDF-sc to GEDF in static systems, with overheads measured from a kernel implementation of EDF-sc. We also show the results of an experimental evaluation of our heuristics using this implementation, 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.

We presented an earlier version of this work in a conference paper (Hobbs et al. 2019). This paper builds upon that work by contributing an open-source kernel implementation of EDF-sc, which was used to conduct new experiments that take real-world scheduling overheads into account. New heuristics are introduced for this implementation, as we found some heuristics from the conference paper to be impractical outside the theoretical realm.

1.3 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 (Block et al. 2005), our notion of a dynamic task system is sufficient for many real-world use cases such as *mode changes*, where the set of tasks executing on a system must change due to some change in the system's environment.[1] 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 τ may change over time.[2]

Each task $\tau_i \in \mathcal{T}$ is specified by the parameters (C_i, T_i), where C_i denotes τ_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*[3] of task τ_i is denoted by $U_i = C_i/T_i$, and the *utilization* of a set of tasks S is denoted $U(S) = \sum_{\tau_i \in S} U_i$. 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 jth job of task τ_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 τ_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

[1] Mode change protocols (Real and Crespo 2004; Nélis et al. 2011) have been extensively studied in the real-time literature for both uniprocessor and multiprocessor systems. While EDF-sc could certainly be made to support various types of mode change protocols, they are mentioned here mainly for illustrative purposes, and adding such support is outside the scope of this work.

[2] Because the set τ (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.

[3] 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 (Block et al. 2005, 2008).

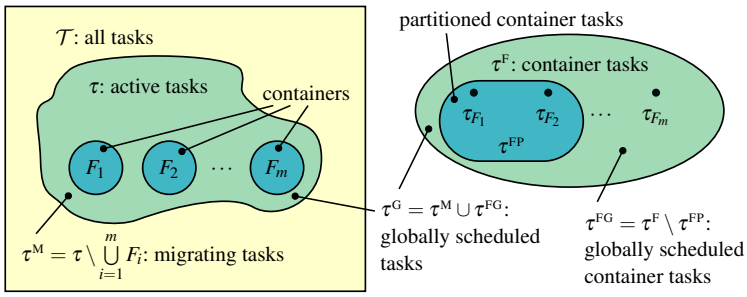


Fig. 1: The relationships between the sets used in EDF-sc.

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 (Devi and Anderson 2008).

3.1 Execution

The EDF-sc scheduling algorithm divides the currently active tasks τ into $m+1$ pairwise-disjoint subsets $F_1, F_2, \ldots, F_m, \tau^{\mathrm{M}}$ such that $\bigcup_{i=1}^{m} F_i \cup \tau^{\mathrm{M}} = \tau$, 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 π_i. Thus, we say that the tasks in F_i are *fixed* on processor π_i, while the tasks in τ^{M} are *migrating*.

We manage the execution budget of each container F_i using a synchronous and periodic *container task* τ_{F_i} with a period T_{F_i} and utilization U_{F_i}. From these two parameters, the container task's execution budget is calculated as $C_{F_i} = U_{F_i} \cdot T_{F_i}$. Unlike the tasks in $\mathcal{T}$, 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 τ_{F_i}. The set of all container tasks is denoted τ^{F}. The period of each container task may be chosen freely and independently by the system designer, so long as $(\forall \tau_{F_i} \in \tau^{\mathrm{F}} :: T_{F_i} > 0)$ holds. The utilizations for all container tasks must be chosen so that the following two conditions are met.

$$(\forall \tau_{F_i} \in \tau^{\mathrm{F}} :: U(F_i) \leq U_{F_i} \leq 1) \tag{1}$$

$$U(\tau^{\mathrm{M}}) + U(\tau^{\mathrm{F}}) \leq m \tag{2}$$

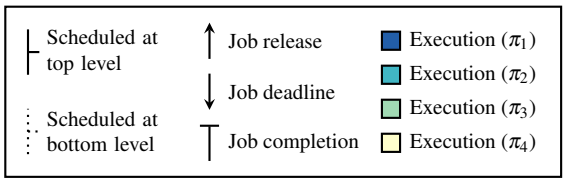


Fig. 2: Legend for Figs. 3, 4, 5, 6, and 7.

If $U_{F_i} = U(F_i)$, then the container task τ_{F_i} is said to be *minimally provisioned*; otherwise, it is *over-provisioned*. If $U_{F_i} = 1$, then τ_{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 $\tau^{\mathrm{FP}} = \{\tau_{F_i} \mid U_{F_i} = 1\}$ (partitioned container tasks) be the set of fully provisioned container tasks. Note that by construction, $U(\tau^{\mathrm{FP}}) = |\tau^{\mathrm{FP}}|$. Also, let $\tau^{\mathrm{FG}} = \{\tau_{F_i} \mid U_{F_i} < 1\}$ (globally scheduled container tasks) be the set of container tasks that are not fully provisioned, and let $\pi^{\mathrm{G}} = \{\pi_i \mid U_{F_i} < 1\}$ 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 $\tau^{\mathrm{G}} = \tau^{\mathrm{M}} \cup \tau^{\mathrm{FG}}$ are scheduled on a GEDF basis on the processors in π^{G}. If a job of a container task τ_{F_i} is selected to run, then it is scheduled on π_i.

S2 Jobs of each container task $\tau_{F_i} \in \tau^{\mathrm{FP}}$ are scheduled on π_i without competition from migrating tasks.

S3 When a container task τ_{F_i} is scheduled, it executes the pending job (if any) of a fixed task $\tau_j \in F_i$ 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 π_i is left idle.

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

Because of its use of containers, 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 $|\pi^{\mathrm{G}}|$ processors. Because F_i always runs on π_i, it may force a job of a migrating task running on π_i 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.

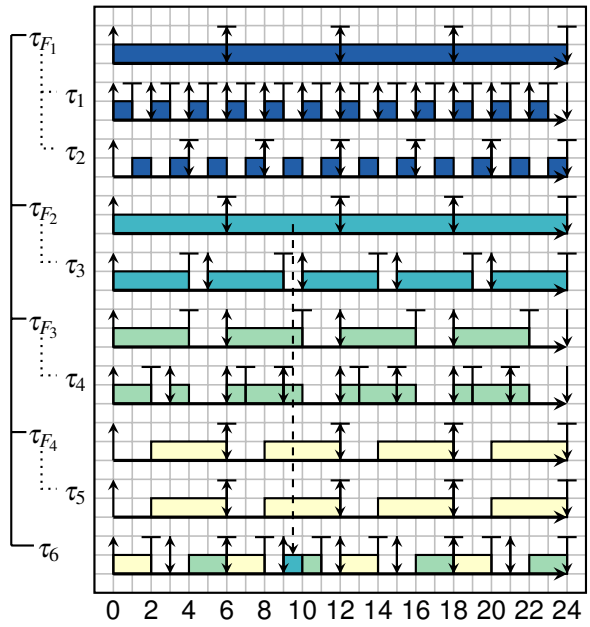


Fig. 3: The schedule described in Example 1.

Example 1 Fig. 3 shows a four-processor EDF-sc schedule of the tasks $\tau_1 = (1,2)$, $\tau_2 = (2,4)$, $\tau_3 = (4,5)$, $\tau_4 = (2,3)$, $\tau_5 = (4,6)$, and $\tau_6 = (2,3)$, partitioned so that $F_1 = \{\tau_1, \tau_2\}$, $F_2 = \{\tau_3\}$, $F_3 = \{\tau_4\}$, $F_4 = \{\tau_5\}$, and $\tau^{\mathrm{M}} = \{\tau_6\}$. All container tasks τ_{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 τ_{F_1} and τ_{F_2} are fully provisioned, so they are scheduled continuously by Rule S2. The other container tasks compete on a GEDF basis with τ_6 by Rule S1. $U(F_2) = 4/5 < U_{F_2}$, so π_2 is idle over several intervals. During one of these, $[9,10)$, jobs of both τ_{F_3} and τ_{F_4} are scheduled by GEDF, but because there are no pending jobs of any task in F_2, τ_6 is scheduled on π_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 (Devi and Anderson 2008). Bounded tardiness for fixed tasks likewise follows from uniprocessor scheduling analysis with limited processor availability (Mok et al. 2001). Thus, for static sporadic task systems, the SRT optimality of EDF-sc (bounded tardiness if no overutilization) 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 τ, as well as changing container weights.[4] 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 $\tau_a \notin \tau$ may be added to τ^{M} if and only if the following condition holds.

$$U_a \leq \min\left(1, \left|\pi^{\mathrm{G}}\right| - U(\tau^{\mathrm{G}})\right) \tag{3}$$

AF (add fixed task) A task $\tau_a \notin \tau$ may be added to the container F_i if and only if the following condition holds.

$$U_a \leq U_{F_i} - U(F_i) \tag{4}$$

R (remove task) A task $\tau_r \in \tau$ whose most recently released job $\tau_{r,i}$ completes at time t_c may be removed from τ 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 τ_{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.

$$U(F_i) \leq U'_{F_i} \leq 1 \tag{5}$$

$$U'_{F_i} - U_{F_i} + U(\tau^{\mathrm{M}} \cup \tau^{\mathrm{F}}) \leq m \tag{6}$$

The weight change only affects newly released jobs of τ_{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 π^{G} 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 (2006). We then use this bound to derive a tardiness bound for fixed tasks under EDF-sc in Sec. 4.2.

[4] Alternative reweighting rules could free system utilization more aggressively than the ones presented here. In particular, a removed migrating task's utilization could be freed at the deadline of its last job. This would allow dynamic workload changes to be made more quickly, but would also create a blocking term in the tardiness analysis for fixed tasks to account for tasks that are being changed from migrating to fixed. To aid in understanding, we opt for more conservative reweighting rules in this work.

4.1 Tardiness Bound for Migrating and Container Tasks

By Rule S1, the migrating and container tasks in EDF-sc compete on a GEDF basis on $|\pi^{\mathrm{G}}|$ 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 (2006) 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* $\tau_1^{\mathrm{c}}, \tau_2^{\mathrm{c}}, \ldots, \tau_n^{\mathrm{c}}$ 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 τ_i^{c} is at most[5]

$$\frac{\sum_{\tau_z^{\mathrm{c}} \in \mathscr{C}^{\mathrm{cmax}}(m-1)} C_z^{\mathrm{cmax}}}{m - \sum_{\tau_z^{\mathrm{c}} \in \mathscr{U}^{\mathrm{cmax}}(m-2)} U_z^{\mathrm{cmax}}} + C_i^{\mathrm{cmax}}, \tag{7}$$

where C_z^{cmax} and U_z^{cmax} are the maximum execution cost and utilization of any task in task class τ_z^{c}, and $\mathscr{C}^{\mathrm{cmax}}(\ell)$ and $\mathscr{U}^{\mathrm{cmax}}(\ell)$ are the subsets of ℓ task classes with the greatest values of C^{cmax} and U^{cmax}, respectively.

Example 2 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, $\tau_1^{\mathrm{c}} = \{\tau_1 : (2,4)\}$, $\tau_2^{\mathrm{c}} = \{\tau_2 : (2,3)\}$, and $\tau_3^{\mathrm{c}} = \{\tau_3 : (5,5), \tau_4 : (4,5)\}$. Before time 10, only tasks τ_1 and τ_3 are active, giving a system utilization of $3/2$. At time 10, τ_3 terminates and τ_2 and τ_4 activate, increasing the system utilization to $59/30$. At time 16, task τ_2 terminates, decreasing the system utilization to $13/10$.

Lemma 1 *Migrating and container tasks in EDF-sc can be modeled by Devi's extended sporadic task model.*

Proof Throughout this proof, we illustrate how Devi's model can be used to model dynamic behaviors by comparing the schedules in Figs. 4 and 5. As there are no inter-task precedence constraints in EDF-sc, each task $\tau_i \in \mathcal{T}$ can be modeled in Devi's model as a task class τ_i^{c}, where each task in τ_i^{c} has the same parameters as τ_i. Rule R in EDF-sc allows a migrating task τ_i to be removed at the deadline or completion time of

[5] New tardiness analysis techniques for GEDF (Erickson et al. 2010; Leoncini et al. 2018) have been proposed since Devi's work, and could likely be applied to obtain reduced bounds for the extended sporadic task model. However, deriving new bounds for existing scheduling algorithms is beyond the scope of this work.

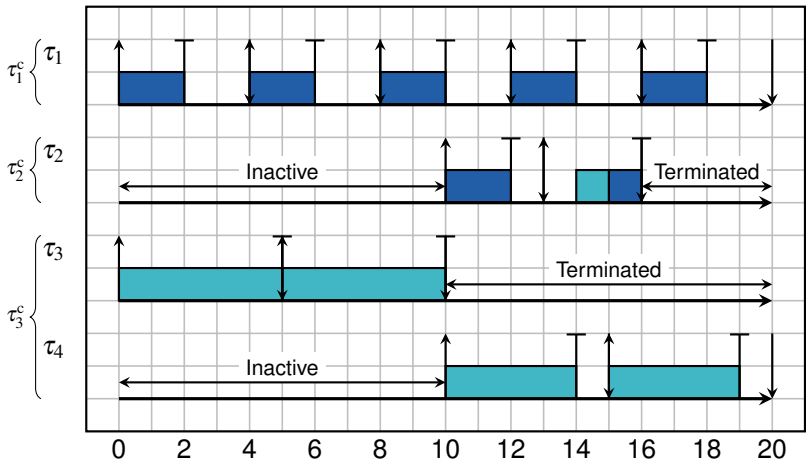


Fig. 4: The extended sporadic task system in Example 2.

its most recently released job, or any time thereafter. This is similar to terminating the active task in τ_i^c in Devi's model, but more conservative because the task's utilization will be reserved after its last released job's deadline if it is tardy. Because Rule AM ensures that the set of tasks scheduled by GEDF never overutilizes the processors on which they are scheduled, adding a task τ_i with Rule AM in EDF-sc is equivalent to activating a task in τ_i^c in Devi's model. These equivalences are shown by task τ_1 in Fig. 5, which is added by Rule AM at time 10 and removed by Rule R at time 16. This is equivalent to τ_2^c under Devi's model in Fig. 4.

Each container task $\tau_{F_i} \in \tau^F$ is likewise equivalent to a task class $\tau_{F_i}^c$ in Devi's model. Initially, each class $\tau_{F_i}^c$ has an active task with parameters (C_{F_i}, T_{F_i}). Each time τ_{F_i} is reweighted with Rule W, the active task in $\tau_{F_i}^c$ terminates, and a task with the new parameters of τ_{F_i} activates. Condition (6) ensures that the new set of active tasks will have total utilization at most m. For example, container task τ_{F_2} in Fig. 5 initially has parameters $(5,5)$, and is reweighted with Rule W at time 10, giving it new parameters $(4,5)$. This task is modeled in Fig. 4 by task class τ_3^c: task $\tau_3 = (5,5)$ is active until time 10, when it terminates and task $\tau_4 = (4,5)$ activates. □

Using Lemma 1 and the tardiness bound (7) for GEDF under the extended sporadic task model, we now derive a tardiness bound for migrating and container tasks in EDF-sc.

Theorem 1 *Tardiness for any migrating or container task τ_i under EDF-sc is at most*

$$\frac{\sum_{\tau_z \in \mathscr{C}^{\max}(\tau^M \cup \tau^F, m-1)} C_z^{\max}}{m - \sum_{\tau_z \in \mathscr{U}^{\max}(\tau^M \cup \tau^F, m-2)} U_z^{\max}} + C_i^{\max}, \tag{8}$$

where $C_z^{\max}$ and $U_z^{\max}$ are the maximum execution cost and utilization of task τ_z, and $\mathscr{C}^{\max}(\tau, \ell)$ and $\mathscr{U}^{\max}(\tau, \ell)$ are the subsets of ℓ tasks in τ with the greatest values of $C^{\max}$ and $U^{\max}$, respectively.

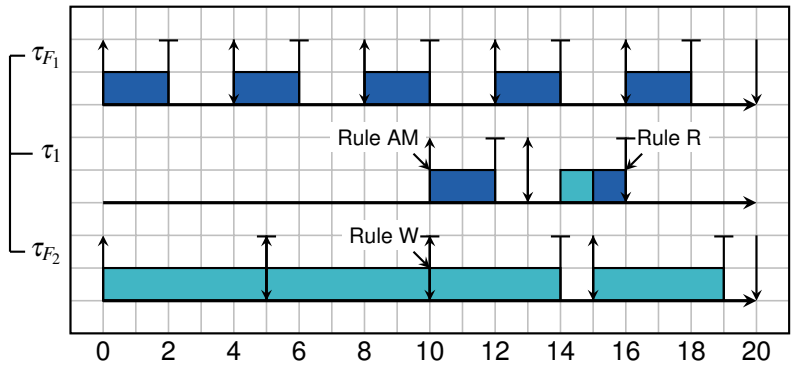


Fig. 5: An EDF-sc schedule analogous to the one in Fig. 4.

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 1.

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 τ^{G} to be scheduled using GEDF on $|\pi^{\mathrm{G}}| < m$ processors. Thus, by Lemma 1, tardiness for a task $\tau_i \in \tau^{\mathrm{G}}$ is upper-bounded by

$$\frac{\sum_{\tau_z \in \mathscr{C}^{\max}(\tau^{\mathrm{G}}, |\pi^{\mathrm{G}}|-1)} C_z^{\max}}{|\pi^{\mathrm{G}}| - \sum_{\tau_z \in \mathscr{U}^{\max}(\tau^{\mathrm{G}}, |\pi^{\mathrm{G}}|-2)} U_z^{\max}} + C_i^{\max}. \tag{9}$$

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 $\tau^{\mathrm{M}} \cup \tau^{\mathrm{F}}$ excluded from the summation in the denominator of (9) are the fully provisioned containers τ^{FP}, whose total maximum utilization is $m - |\tau^{\mathrm{G}}|$. Therefore, tardiness for tasks in τ^{G} is still upper-bounded by (8), so the theorem holds. □

The tardiness bound in Theorem 1 may be impractical to compute directly, as it requires knowledge of the online behavior of the system, which may not be known in advance. However, by assuming the worst case where each container task may have utilization 1 and the tasks in $\mathcal{T}$ with the highest execution cost are at some point in τ^{M}, we obtain the following bound, which can be computed offline.

Corollary 1 *Tardiness for any migrating or container task τ_i under EDF-sc is at most*

$$\frac{\sum_{\tau_z \in \mathscr{C}^{\max}(\mathcal{T} \cup \tau^{\mathrm{F}}, m-1)} C_z^{\max}}{2} + C_i^{\max}, \tag{10}$$

where

$$C_z^{\max} = \begin{cases} C_z & \text{if } \tau_z \in \mathcal{T} \\ T_z & \text{if } \tau_z \in \tau^{\mathrm{F}}, \end{cases}$$

and $\mathscr{C}^{\max}(\tau, \ell)$ is the subset of ℓ tasks in τ with the greatest value of $C^{\max}$.

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 (Leontyev et al. 2011; Mok et al. 2001), 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 2 *Under EDF-sc, tardiness for fixed tasks on processor π_i is at most*

$$\sigma_i = 2T_{F_i} - 2C_{F_i}^{\min} + \frac{\sum_{\tau_z \in \mathscr{C}^{\max}(\tau^{\mathrm{M}} \cup \tau^{\mathrm{F}}, m-1)} C_z^{\max}}{m - \sum_{\tau_z \in \mathscr{U}^{\max}(\tau^{\mathrm{M}} \cup \tau^{\mathrm{F}}, m-2)} U_z^{\max}} + C_{F_i}^{\max}, \tag{11}$$

where $C_z^{\max}$, $U_z^{\max}$, $\mathscr{C}^{\max}(\tau, \ell)$, and $\mathscr{U}^{\max}(\tau, \ell)$ are as in Theorem 1, and $C_{F_i}^{\min}$ is the lowest budget assigned to τ_{F_i} for any t where $F_i(t) \neq \emptyset$.

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

We next derive bounds for processor demand and supply from time t_0 to an arbitrary time $u > t_0$, where for every $t \in [t_0, u)$, $U(F_i(t)) > 0$ holds. Such an interval $[t_0, u)$ is called *non-empty*. Because the interval of interest $[t_0, t_c)$ is non-empty, we can then use these bounds to derive a completion time for $\tau_{a,b}$. 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. To allow for tighter accounting, we define the set $\widehat{F}_i(t)$ by

$$\widehat{F}_i(t) = \left\{ \tau_h \mid \left(\exists j : t_0 \leq r_{h,j} \leq t :: \left(\forall s : r_{h,j} \leq s \leq t :: \tau_h \in F_i(s) \right) \right) \right\},$$

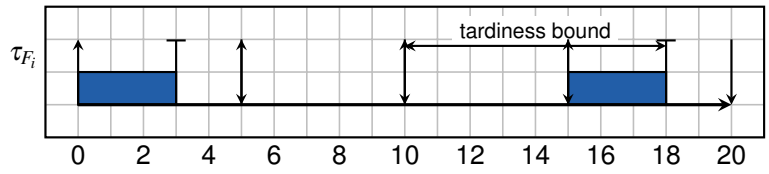


Fig. 6: An example of the maximum duration of processor unavailability in the proof of Theorem 2.

which intuitively considers each task τ_h as having been added by Rule AF at the moment of its first job release after it was actually added, or after t_0 if $\tau_h \in F_i(t_0)$. Using this definition, the processor demand created by jobs of fixed tasks on π_i over an interval $[t_0, u)$ is at most

$$\alpha_i(u) = \int_{t_0}^{u} U\left(\widehat{F}_i(t)\right)\, dt.$$

Therefore, the processor demand created by competing jobs released at or after t_0 is at most $\alpha_i(d_{a,b})$.

Now we derive a lower bound to processor availability to fixed tasks on π_i. Prior work involving limited processor availability for static systems typically provides a lower bound for an interval of length Δ of the form

$$\beta(\Delta) = \max\left(0, \widehat{U} \cdot (\Delta - \sigma)\right),$$

where $\widehat{U}$ is the long-term average processor availability, and σ is the maximum duration of time where the processor can be continuously unavailable (Leontyev et al. 2011). 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 a non-empty interval $[t_0, u)$ that takes dynamic behavior into account.

To determine the maximum duration of processor unavailability, consider the example in Fig. 6. In the figure, the container task τ_{F_i} is assumed to have a period of 5 and a budget of 3, while σ_i is 8. The first job $\tau_{F_i,1}$ runs as soon as it is released, completing at time 3; the second job $\tau_{F_i,2}$ completes just at its tardiness bound, beginning its execution at time 15. As it is not possible for these jobs to be spaced any farther apart, the maximum processor unavailability in this case is 12. In general, we can see that the maximum processor unavailability occurs when a job $\tau_{F_i,j}$ completes execution as early as possible, and the next job $\tau_{F_i,j+1}$ completes as late as possible. Thus by adding two container periods, less two minimum container task budgets, to the tardiness bound from Theorem 1, we arrive at σ_i from the theorem statement as an upper bound to the maximum duration of processor unavailability regardless the container task's utilization.

We now show that the processor availability over any non-empty interval $[t_0, u)$ is at least

$$\beta_i(u) = \max\left(0, \int_{t_0}^{u-\sigma_i} U\left(\widehat{F}_i(t)\right)\, dt\right).$$

The reweighting rules in Sec. 3.2 guarantee that at any time t, $U(F_i(t)) \leq U_{F_i}(t)$. By construction, $\widehat{F}_i(t) \subseteq F_i(t)$ holds for any time t, so it is safe to use the utilization of the contained tasks $U(\widehat{F}_i(t))$ in a lower bound for processor availability. The function $\beta_i(u)$ assumes that the processor is initially unavailable for σ_i time units. Because this occurs when a job $\tau_{F_i,j}$ completes as late as possible, the entire budget of $\tau_{F_i,j}$ must then be consumed non-preemptively. Furthermore, to ensure the tardiness bound is met, each subsequent job $\tau_{F_i,j+k}$ where $k > 0$ must now consume its budget over an interval of length at most T_{F_i}, maintaining an average availability of $U_{F_i}(r_{F_i,j+k})$ 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, $\alpha_i(d_{a,b})$, equals the minimum processor supply from t_0 to $d_{a,b} + \sigma_i$, $\beta_i(d_{a,b} + \sigma_i)$. Therefore, all work created by competing jobs must have completed by time $d_{a,b} + \sigma_i$. This contradicts the assumption that $\tau_{a,b}$ completes at time $t_c > d_{a,b} + \sigma_i$, so the theorem holds. □

As in Sec. 4.1, the bound in Theorem 2 requires knowledge of the system's online behavior. We can likewise obtain the following tardiness bound that can be computed offline by applying Corollary 1 to upper-bound the tardiness of τ_{F_i}, and by computing the minimum budget of τ_{F_i} from the lowest utilization of any task in $\mathcal{T}$.

Corollary 2 *Under EDF-sc, tardiness for fixed tasks on processor π_i is at most*

$$\sigma_i = 3T_{F_i} - 2U_{\min}T_{F_i} + \frac{\sum_{\tau_z \in \mathscr{C}^{\max}(\mathcal{T} \cup \tau^{\mathrm{F}}, m-1)} C_z^{\max}}{2}, \tag{12}$$

where $U_{\min}$ is the lowest utilization of any task in $\mathcal{T}$.

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 τ 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 (Devi 2006).

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 task 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 container tasks 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 τ 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* (Ivkovic and Lloyd 1998). 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.

5.2.1 Task addition

Requests to add tasks to τ may be issued at any time using the procedure ADDTASK(τ_a). 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 ADDTASK procedure simply adds the task τ_a 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 CONTAINERSELECTIONHEURISTIC for each task in FIFO order to attempt to add tasks into temporary containers $F'_1, \ldots, F'_m, \tau^{M\prime}$. This heuristic uses a bin-packing heuristic to find a container F'_i so that $U(F'_i) + U_a \leq 1$ holds. 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_a can be added as a migrating task by checking if

Algorithm 1 Heuristics for adding and removing tasks, and for stabilizing the workload.

PendingAdds: FIFO queue, initially empty

1: **procedure** ADDTASK(τ_a)
2: PendingAdds.ENQUEUE(τ_a)

3: **procedure** REMOVETASK(τ_r)
4: **if** τ_r's most recently released job $\tau_{r,j}$ is pending **then**
5: Forbid τ_r from releasing new jobs
6: Schedule R(τ_r) to occur at the later of $d_{r,j}$ and $\tau_{r,j}$'s completion time
7: **else**
8: R(τ_r) ▷ Remove the task now

9: **procedure** CONTAINERBOUNDARY()
▷ Function called at each container job boundary t
10: $F'_1, \ldots, F'_m, \tau^{M\prime} \leftarrow F_1, \ldots, F_m, \tau^M$
▷ Determine which tasks can be added
11: **while** PendingAdds is not empty **do**
12: $\tau_a \leftarrow$ PendingAdds.DEQUEUE()
13: Select container F'_i, $\tau^{M\prime}$, or $\perp$ for τ_a via CONTAINERSELECTIONHEURISTIC
14: Add τ_a to selected container, or reject if $\perp$
▷ Try to move migrating tasks into containers
15: **for each** $\tau_k \in \tau^M$ **do**
16: $\tau'_k \leftarrow \tau_k$
17: **if** τ_k's most recently released job $\tau_{k,j}$ is not pending **and** $d_{k,j} < t + T_{F_1}$
and τ'_k will fit in some container F'_i **then**
18: Select container F'_i for τ'_k via CONTAINERSELECTIONHEURISTIC
19: Add τ'_k to selected container
20: Schedule an atomic operation $\langle$R(τ'_k); R(τ_k); AF(τ_k, F_i)$\rangle$ to occur at
$\max(d_{k,j}, t)$, i.e., use Rules R and AF to enact the move of τ_k into F_i
21: Compute new container task weights via CONTAINERREWEIGHTINGHEURISTIC
22: Reweight container tasks using Rule W
▷ Enact all non-rejected task additions using Rules AF and AM
23: **for** $i \leftarrow 1 \ldots m$ **do**
24: **for each** $\tau_k \in F'_i \setminus F_i$ **do**
25: AF(τ_k, F_i)
26: **for each** $\tau_k \in \tau^{M\prime} \setminus \tau^M$ **do**
27: AM(τ_k)

$\sum_{i=1}^{m} U(F'_i) + U(\tau^{M\prime}) + U_a \leq m$ holds. If not, the heuristic returns $\perp$, and τ_a 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.

5.2.2 Task removal

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

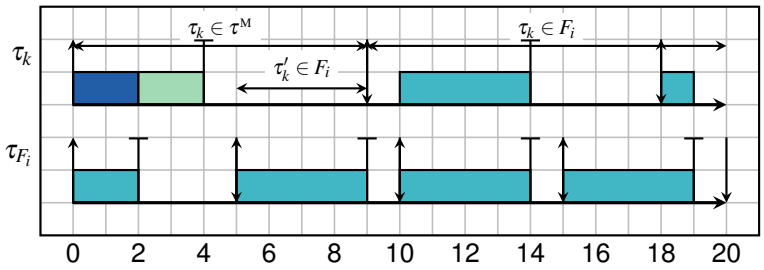


Fig. 7: Task τ_k is eligible to be moved at time 5, since $\tau_{k,1}$ completed at time 4, and $d_{k,1} = 9 < d_{F_i,2} = 10$. Its copy τ'_k is added to F_i at time 5, and at time 9 the move is completed. Such "copy" tasks merely reserve container capacity and are not executed.

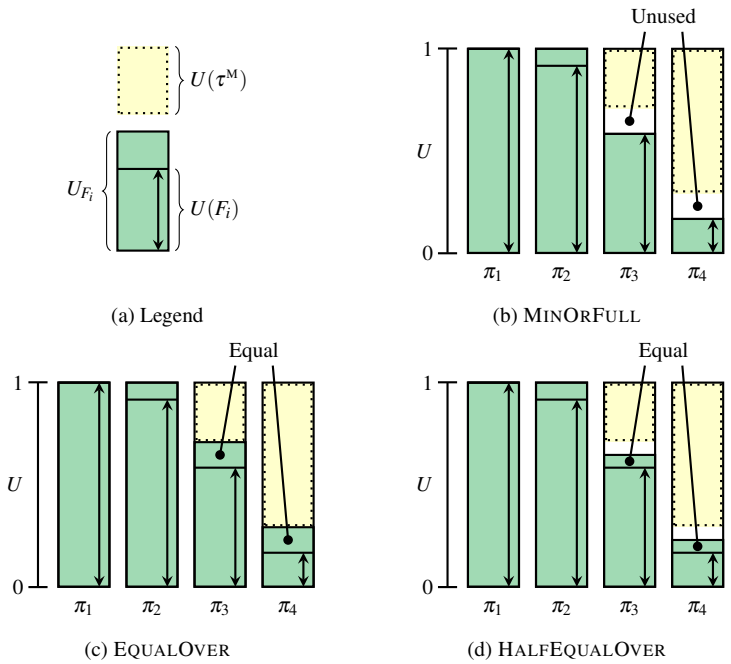
5.2.3 Stabilization

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. 7. Using the rules from Sec. 3.2, this must be done by removing a migrating task τ_k 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 $\tau_{k,j}$ completes at a future time after its deadline, so to avoid this, we require that $\tau_{k,j}$ 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 by creating a copy of τ_k at line 16, denoted τ'_k, 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 to that container at line 19. We then schedule an atomic operation at line 20 to remove τ'_k from the container, remove τ_k as a migrating task, and add τ_k to the container at the deadline of its most recently released job.

Note that if the task system has not changed since the prior invocation of the ContainerBoundary procedure, there is no need to run lines 15–20, since the stabilization routine will again find no migrating tasks to move into containers. Our kernel implementation of EDF-sc, as discussed in Sec. 6, makes use of this optimization.



(a) Legend (b) MINORFULL

(c) EQUALOVER (d) HALFEQUALOVER

Fig. 8: The three container task provisioning techniques discussed. **(b)** Container tasks τ_{F_3} and τ_{F_4} are minimally provisioned ($U_{F_i} = U(F_i)$), and τ_{F_1} and τ_{F_2} are fully provisioned ($U_{F_i} = 1$). **(c)** Container task utilizations U_{F_3} and U_{F_4} are increased equally to consume the system capacity left unused by MINORFULL. **(d)** Container task utilizations U_{F_3} and U_{F_4} are increased by half as much as with EQUALOVER to leave some extra capacity for migrating tasks.

5.2.4 Container task reweighting

Once all operations that add tasks to containers have been planned, we call a CONTAINERREWEIGHTINGHEURISTIC at line 21 to determine new container weights based on the sets $F'_1, \ldots, F'_m, \tau^{M\prime}$. This procedure determines new weights for the container tasks so that for each τ_{F_i}, $U(F'_i) \leq U_{F_i} \leq 1$ holds, and so that $\Sigma_{i=1}^{m} U_{F_i} + U(\tau^{M\prime}) \leq m$ holds. Once the new weights have been determined, the container tasks are all reweighted using Rule W at line 22. As with stabilization, this can be skipped if there have been no changes to the task system since the last invocation of CONTAINERBOUNDARY, as the container task weights need not be changed. There are many different ways to determine new container task weights; in prior work (Hobbs et al. 2019) we proposed the following two techniques.

MINORFULL (Fig. 8b) Begin by minimally provisioning all container tasks. Then fully provision as many as possible while maintaining $U(\tau^{G}) \leq |\pi^{G}|$, e.g. by choosing container tasks in order of decreasing $U(F_i)$. This may leave some processor capacity in π^{G} unavailable to fixed tasks.

EQUALOVER (Fig. 8c) Fully provision container tasks as with MINORFULL, but over-provision each of the other $|\pi^{\mathrm{G}}|$ container tasks by $(m - U(\tau))/|\pi^{\mathrm{G}}|$.

Unfortunately, neither of these heuristics can be used directly in a real implementation of EDF-sc. Scheduler overheads would cause MINORFULL to effectively under-provision container tasks, resuling in unbounded tardiness for fixed tasks. EQUALOVER would likewise reserve insufficient system utilization for migrating tasks when overheads are considered, violating the assumptions needed for the tardiness bound proven in Theorem 1. To address this issue, we propose one further heuristic.

HALFEQUALOVER (Fig. 8d) Fully provision container tasks as with MINORFULL, but over-provision each of the other $|\pi^{\mathrm{G}}|$ container tasks by $(m - U(\tau))/(2 \cdot |\pi^{\mathrm{G}}|)$.

This new heuristic is a simple modification of EQUALOVER that assigns half the total remaining capacity to migrating tasks, and the other half to be split among the container tasks. This ensures that both fixed and migrating tasks have extra processor capacity for scheduler overhead, so tardiness will not grow without bound.

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 begin by demonstrating the advantages of semi-partitioned scheduling over global scheduling with an overhead-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 (Hobbs et al. 2020).

6.1 Overhead-Aware Schedulability Study

To illustrate the advantages of semi-partitioned scheduling over global scheduling, we performed an overhead-aware schedulability study comparing EDF-sc to GEDF with a variety of static task systems. In order to accurately compare these schedulers in a practical setting, we produced a kernel implementation of EDF-sc in LITMUS$^{\mathrm{RT}}$ version 2017.1 (Calandrino et al. 2006; Brandenburg 2011), based on the Linux kernel version 4.9.30. Although this version of LITMUS$^{\mathrm{RT}}$ provides support for *reservations* that are similar to the containers used in EDF-sc, these reservations are unfortunately not suitable for GEDF scheduling, so our implementation handles container and fixed tasks manually. This open-source implementation is available online (Hobbs et al. 2020).

Table 1: Average case overheads of EDF-sc for different task set sizes.

TASKS	CXS (μs)	RELEASE LATENCY (μs)	RELEASE (μs)	CONTAINER BOUNDARY (μs)	SCHED (μs)	SCHED2 (μs)	SEND RESCHED (μs)
24	2.220	20.027	0.894	54.559	4.544	0.163	2.234
48	2.234	12.563	1.018	55.526	4.693	0.173	2.213
72	1.777	9.974	0.887	54.442	4.152	0.177	2.123
96	1.371	1.758	1.018	56.076	3.470	0.180	2.106
120	1.288	0.797	1.177	63.467	3.498	0.166	2.133
144	1.250	0.759	1.188	65.212	3.321	0.166	2.133
168	1.216	0.832	1.174	60.320	3.030	0.166	2.123
192	1.195	0.790	1.236	59.661	2.992	0.156	2.123
216	1.281	0.790	1.365	58.048	3.855	0.166	2.043
240	1.368	0.786	1.344	59.325	3.875	0.156	2.040

Table 2: Average case overheads of GEDF for different task set sizes.

TASKS	CXS (μs)	RELEASE LATENCY (μs)	RELEASE (μs)	SCHED (μs)	SCHED2 (μs)	SEND RESCHED (μs)
24	1.070	15.749	2.334	9.825	0.166	2.320
48	1.423	11.263	2.376	20.606	0.177	2.282
72	1.808	8.083	2.386	18.083	0.170	2.234
96	2.210	1.282	2.303	11.574	0.159	2.123
120	2.241	0.882	2.210	9.881	0.149	2.043
144	2.248	0.873	2.210	9.818	0.149	2.033
168	2.258	0.935	2.140	10.552	0.149	2.054
192	2.261	0.930	2.192	10.348	0.149	2.040
216	2.265	0.884	2.140	10.494	0.149	2.012
240	2.279	0.914	2.154	10.677	0.149	2.023

We measured overheads of EDF-sc and GEDF following the methodology of Bastoni et al. (2011), additionally measuring the execution time of the CONTAINERBOUNDARY routine in EDF-sc. We considered the average case overheads for these experiments, since we are considering SRT systems, which are more likely to be provisioned based on the average case. Tables 1 and 2 show the overheads of EDF-sc and GEDF, respectively, for task sets of size $24, 48, 72, \ldots, 240$. In these tables, CXS denotes the overhead of context switches. RELEASE LATENCY is the duration of time between a job's actual release and the moment its release timer signals for its release. RELEASE is the overhead of releasing the job after its release timer expires. SCHED and SCHED2 denote the overhead of each scheduling decision and its post decision bookkeeping, respectively. SEND RESCHED is the inter-processor interrupt overhead necessary for global scheduling. The overhead of the CONTAINERBOUNDARY routine in EDF-sc is given, and is absent for GEDF, which has no equivalent routine.

Notably, EDF-sc has values of SCHED noticeably lower than those of GEDF. Moreover, lower values were observed in systems with more tasks. Intuitively, this effect is due to the semi-partitioned nature of EDF-sc. Most tasks in EDF-sc are packed into containers, and thus multiple per-core run queues, which reduces the size of the global run queue. When the number of scheduled tasks in a system increases, the

likelihood of each task having lower utilization also increases. This results in a more efficient packing in EDF-sc, and thus more fully-provisioned containers, for which the global run queue need not be considered when making scheduling decisions.

In these experiments, we randomly generated static sporadic task systems and increased the execution times of each task to account for average-case overheads on a 24-core Intel system. This is the same system used by Bastoni et al. (2010), so the same cache-related preemption and migration delay (CPMD) values were used in our experiments. These CPMD values included measurements for both idle systems (low cache contention, where all tasks running on the system were real-time) and systems under load (high cache contention, where best-effort cache thrashing tasks were also run). After inflating the execution time, we calculated the *schedulability*, or proportion of the generated task systems that remain feasible, under each scheduler. Task sets were generated similarly to Bastoni et al. (2011), 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 $[10, 24]$ 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 HALFEQUALOVER 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. To aid in the presentation of a large number of schedulability experiments, we use *weighted schedulability* (Bastoni et al. 2010), defined as

$$S(W) = \frac{\sum_{U \in Q} U \cdot S(U, W)}{\sum_{U \in Q} U},$$

where Q is the set of utilization caps considered, W is a WSS value, and $S(U, W)$ is the schedulability ratio for utilization cap U and WSS W. Using weighted schedulability collapses an entire traditional schedulability plot into a single point, allowing us to explore the parameter space without resorting to 3D plots.

Results. We discuss a representative subset of our schedulability experiments here; the full results are given in the appendix. Typical results are shown in Fig. 9 for task sets with uniform medium utilizations. As shown in the left subfigure, when idle

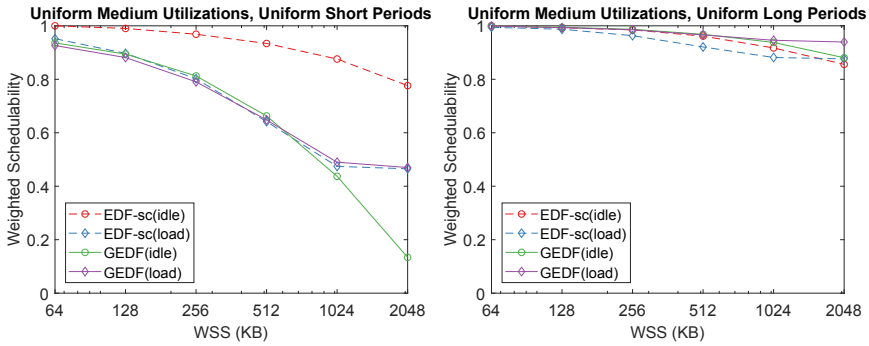


Fig. 9: Left: weighted schedulability for task sets with uniform medium utilizations and uniform short periods. Right: weighted schedulability for task sets with uniform medium utilizations and uniform long periods.

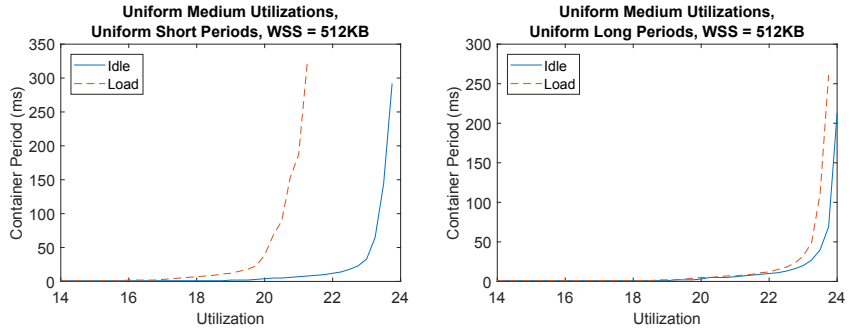


Fig. 10: Minimum container periods required to schedule medium-utilization task sets generated with a uniform distribution and differing periods at WSS = 512.

CPMD overheads are considered, EDF-sc performs significantly better than GEDF for task systems with short periods. Under load overheads, EDF-sc performs slightly worse than or comparably to GEDF. This behavior is expected, as the CPMD overhead for migrations through L1 cache is similar to a memory level migration under intense memory traffic. Therefore, the advantage of semi-partitioning allowing fixed tasks to only experience L1 migrations is nullified. Due to several effects influencing the measurement of CPMD values, GEDF gave higher schedulability with large WSS values when using load overheads instead of idle overheads; please see Bastoni et al. (2010) for more details. Systems with bimodal and exponential utilization distributions showed similar trends. GEDF is more competitive when task periods are longer, and in some cases, can achieve marginally higher weighted schedulability than EDF-sc.

Fig. 10 shows the averages of the minimum container periods for which the generated task systems remained schedulable under EDF-sc. The minimum feasible container task periods remained very low (typically under about 50 ms) for most of the

feasible task sets, increasing sharply for higher utilizations where most task sets were unschedulable. This trend holds under both idle and load CPMD overheads. Since fewer task systems are schedulable with load overheads, the sharp increase in container task period occurs at a lower utilization. The full set of plots for these experiments is 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 24-processor machine running a working EDF-sc implementation with 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. (2017). Each dynamic workload consisted of a task set $\mathcal{T}$, and a sequence of task addition and removal events. The tasks in $\mathcal{T}$ 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 μ_U and variance σ_U^2. We used values for μ_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 σ_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 τ in each dynamic workload consisted of tasks from $\mathcal{T}$ chosen at random one at a time until the next chosen task would have become unschedulable by the overhead-aware schedulability test used in Sec. 6.1. Each workload was subsequently modified by a sequence of 25 task addition or removal events over the course of the workload execution, with event 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 $\Lambda = (1 - U(\tau)/m) + \psi(U(\tau)/m)$ 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 $\mathcal{T} \setminus \tau$ to τ. Otherwise, the event selected a random task to be removed from τ. 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 (Casini et al. 2017). We held $\psi = 0.8$ across all generated workloads to keep system utilization high.

For each combination of μ_U value and WSS for each power of two in $[64, 2048]$ kB, we generated 50 dynamic workloads. Each workload is scheduled on a 24-processor

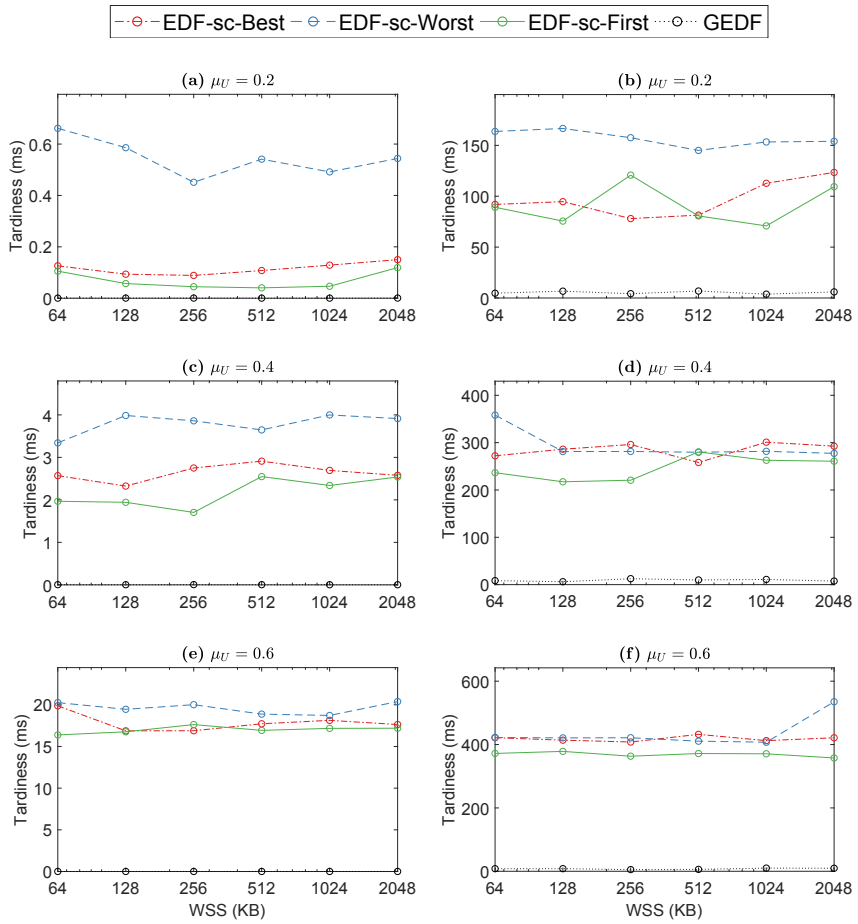


Fig. 11: Insets (a), (c), and (e) (left) show observed average tardiness over easy-, moderate-, and hard-to-partition workloads. Insets (b), (d), and (f) (right) show observed maximum tardiness over easy-, moderate-, and hard-to-partition workloads.

system using EDF-sc, with each combination of the bin-packing heuristics BESTFIT, WORSTFIT, and FIRSTFIT, and the HALFEQUALOVER container reweighting heuristic. Following the results of the experiments in Sec. 6.1, we used container periods of 50 ms in our implementation. We also scheduled the same dynamic workloads under GEDF to compare tardiness between the two algorithms.

Results. Fig. 11 shows average and maximum tardiness across all sets of generated workloads that are easy-, moderate-, and hard-to-partition. Plots of the mean of the analytical tardiness bounds computed for these workloads using Corollaries 1 and 2 are available in the appendix. In each graph, the *x* axis shows each WSS considered, and the *y* axis shows the tardiness observed across the generated workloads. A lower

curve represents a better-performing heuristic. To reduce visual clutter, standard deviations are omitted from the figure.

For easy-to-partition task sets, all EDF-sc heuristics gave low average tardiness under 1 ms, but this is still higher than GEDF. In moderately-hard-to-partition and hard-to-partition workloads however, GEDF significantly outperformed EDF-sc on the basis of tardiness. This is expected, since when a large number of tasks have utilization greater than $1/2$, bin-packing will result in many migrating tasks, whose presence significantly decreases the number of fully provisioned containers. The execution of fixed tasks can then be delayed for prolonged periods of time when the containers that are not fully-provisioned run out of budget. It is worth noting that the task set generation yields an expected mean period of 505 ms, so the average tardiness under EDF-sc is still quite low compared to most task periods. It may further be noted that GEDF gave significantly lower tardiness in these real-world experiments than it did in our prior simulated experiments (Hobbs et al. 2019). This is likely due to migrations incurring lower cache-related delays in practice than we had assumed for the simulation.

The maximum tardiness observed for EDF-sc is significantly higher than that for GEDF in these experiments. This is partly because of the delays due to container budget exhaustion, but also due to an inherent tendency of EDF-sc to run the top-level GEDF scheduling with high-utilization container tasks. Tasks with high utilization are known to cause greater actual tardiness and tardiness bounds under GEDF scheduling. Indeed, the mean tardiness bounds for EDF-sc in these task systems were 3–4 times those of GEDF.

The WORSTFIT bin-packing heuristic tended to give higher average and maximum tardiness than BESTFIT and FIRSTFIT. This is likely due to WORSTFIT's tendency to keep the utilization of fixed tasks on each processor balanced, which may in turn make the container reweighting heuristic unable to fully provision many containers. BESTFIT and FIRSTFIT were fairly competitive with one another, but FIRSTFIT gave slightly lower average tardiness for moderate- and easy-to-partition workloads where EDF-sc is most practical. Further, FIRSTFIT has slightly lower runtime overhead, so it may be preferred for this reason as well.

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 HALFEQUALOVER 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. 12. 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

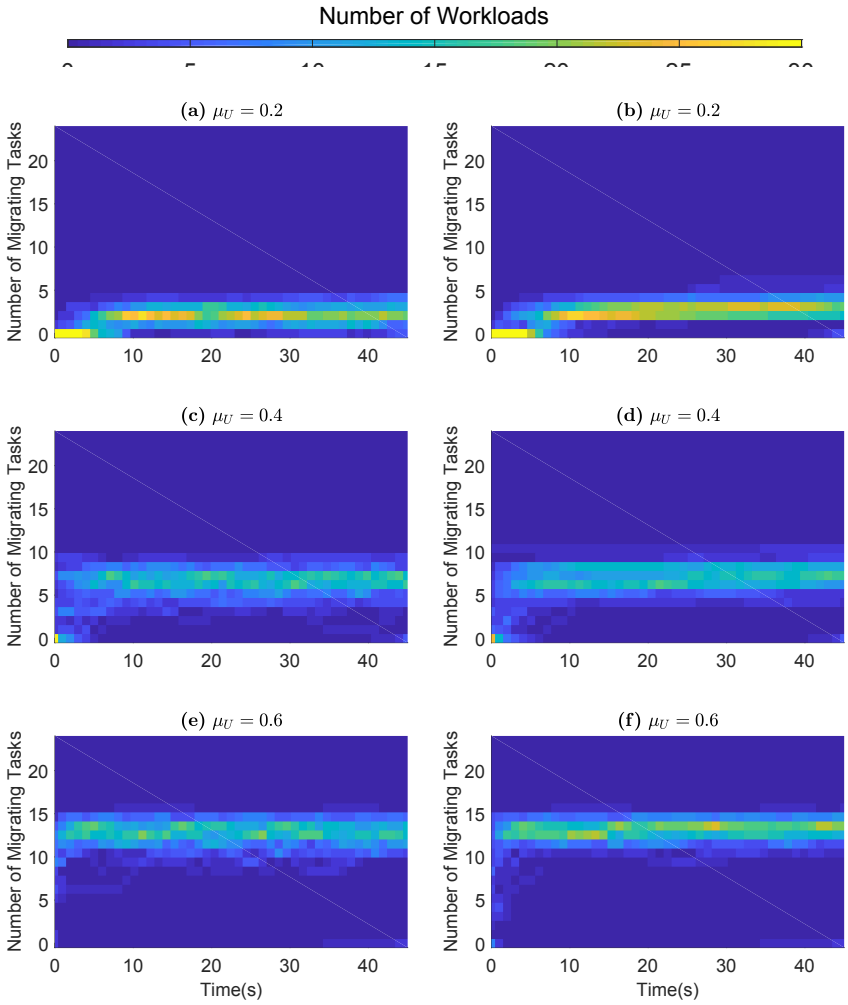


Fig. 12: Stabilization heatmaps for tasks with WSS of 2048 kB. Insets (a), (c), and (e) (left) show easy-, moderate-, and hard-to-partition workloads with stabilization enabled. Insets (b), (d), and (f) (right) show easy-, moderate-, and hard-to-partition workloads with stabilization disabled.

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 45 s in all cases, so we limited the x axis to show the interval $[0,45]$ s. The color of each cell indicates how many workloads had a particular number of migrating tasks at each time observed.

We found that for hard-to-partition workloads, the stabilization code had little effect on the number of migrating tasks. Indeed, for task sets with a mean task uti-

lization of 0.6, it would not be possible in many cases to pack two tasks into a single container. The benefit of stabilization can be seen for the easy-to-partition workloads however, becoming most apparent at $t = 25$. We found that most easy-to-partition workloads remain at 2 migrating tasks instead of increasing to 3 when stabilization is disabled. For moderately-hard-to-partition workloads, the band on the heatmap for the stabilized system is much wider, including many more instances with less than 6 migrating tasks in the system compared to the unstabilized system. This confirms that 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. Finally, we presented the results of experiments performed using a newly-written implementation of EDF-sc in the LITMUS$^{\mathrm{RT}}$ kernel, which show that it is especially competitive with GEDF for task systems with shorter periods or lower per-task utilizations.

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

Anderson JH, Bud V, Devi UC (2005) An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In: Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)

Anderson JH, Erickson JP, Devi UC, Casses BN (2016) Optimal semi-partitioned scheduling in soft real-time systems. Journal of Signal Processing Systems 84(1):3–23

Andersson B, Tovar E (2006) Multiprocessor scheduling with few preemptions. In: Proceedings 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp 322–334

Andersson B, Bletsas K, Baruah SK (2008) Scheduling arbitrary-deadline sporadic task systems on multiprocessors. In: Proceedings of the 29th IEEE Real-Time Systems Symposium (RTSS), pp 385–394

Bastoni A, Brandenburg BB, Anderson JH (2010) Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT) pp 33–44

Bastoni A, Brandenburg BB, Anderson JH (2011) Is semi-partitioned scheduling practical? In: Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS), IEEE, pp 125–135

Bhatti MK, Belleudy C, Auguin M (2012) A semi-partitioned real-time scheduling approach for periodic task systems on multicore platforms. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC), pp 1594–1601

Bletsas K, Andersson B (2009) Notional processors: an approach for multiprocessor scheduling. In: Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 3–12

Bletsas K, Andersson B (2011) Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. Real-Time Systems 47(4):319–355

Block A, Anderson JH, Bishop G (2005) Fine-grained task reweighting on multiprocessors. In: Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp 429–435

Block A, Anderson JH, Devi UC (2008) Task reweighting under global scheduling on multiprocessors. Real-Time Systems 39(1-3):123–167

Brandenburg BB (2011) Scheduling and locking in multiprocessor real-time operating systems. PhD thesis, University of North Carolina at Chapel Hill

Brandenburg BB, Gül M (2016) Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations. In: Proceedings of the 37th IEEE Real-Time Systems Symposium (RTSS), pp 99–110

Burns A, Davis RI, Wang P, Zhang F (2012) Partitioned EDF scheduling for multiprocessors using a C = D task splitting scheme. Real-Time Systems 48(1):3–33

Calandrino JM, Leontyev H, Block A, Devi UC, Anderson JH (2006) LitmusRT: A testbed for empirically comparing real-time multiprocessor schedulers pp 111–123

Casini D, Biondi A, Buttazzo G (2017) Semi-partitioned scheduling of dynamic real-time workload: A practical approach based on analysis-driven load balancing. In: Proceedings of the 29th Euromicro Conference on Real-Time Systems (ECRTS), pp 13:1–13:23

Devi UC (2006) Soft real-time scheduling on multiprocessors. PhD thesis, University of North Carolina at Chapel Hill

Devi UC, Anderson JH (2008) Tardiness bounds under global EDF scheduling on a multiprocessor. Real-Time Systems 38(2):133–189

Dorin F, Yomsi PM, Goossens J, Richard P (2010) Semi-partitioned hard real-time scheduling with restricted migrations upon identical multiprocessor platforms. CoRR abs/1006.2637, URL http://arxiv.org/abs/1006.2637, 1006.2637

Erickson JP, Devi UC, Baruah SK (2010) Improved tardiness bounds for global EDF. In: Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS), pp 14–23

Fan M, Quan G (2012) Harmonic semi-partitioned scheduling for fixed-priority real-time tasks on multicore platform. In: Proceedings of the 2012 Design, Automation Test in Europe Conference Exhibition (DATE), pp 503–508

Guan N, Stigge M, Yi W, Yu G (2010a) Fixed-priority multiprocessor scheduling: Beyond Liu & Layland utilization bound. In: Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS) Work-in-Progress Session, pp 1594–1601

Guan N, Stigge M, Yi W, Yu G (2010b) Fixed-priority multiprocessor scheduling with Liu and Layland's utilization bound. In: Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 165–174

Hobbs C, Tong Z, Anderson JH (2019) Optimal soft real-time semi-partitioned scheduling made simple (and dynamic). In: Proceedings of the 27th International Conference on Real-Time Networks and Systems (RTNS), pp 112–122

Hobbs C, Tong Z, Bakita J, Anderson JH (2020) Statically optimal dynamic soft real-time semi-partitioned scheduling, additional materials. URL http://cs.unc.edu/%7Eanderson/papers.html

Ivkovic Z, Lloyd EL (1998) Fully dynamic algorithms for bin packing: Being (mostly) myopic helps. SIAM Journal on Computing 28(2):574–38

Kato S, Yamasaki N (2007) Real-time scheduling with task splitting on multiprocessors. In: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp 441–450

Kato S, Yamasaki N (2008) Portioned EDF-based scheduling on multiprocessors. In: Proceedings of the 8th ACM International Conference on Embedded Software (EMSOFT), pp 139–148

Kato S, Yamasaki N (2009) Semi-partitioned fixed-priority scheduling on multiprocessors. In: Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 23–32

Leoncini M, Montangero M, Valente P (2018) A parallel branch-and-bound algorithm to compute a tighter tardiness bound for preemptive global EDF. Real-Time Systems pp 1–38, URL https://doi.org/10.1007/s11241-018-9319-6

Leontyev H, Chakraborty S, Anderson JH (2011) Multiprocessor extensions to real-time calculus. Real-Time Systems 47(6):562

Mok AK, Feng X, Chen D (2001) Resource partition for real-time systems. In: Proceedings of the Seventh IEEE Real-Time Technology and Applications Symposium (RTAS), pp 75–84

Nélis V, Andersson B, Marinho J, Petters SM (2011) Global-edf scheduling of multimode real-time systems considering mode independent tasks. In: 2011 23rd Euromicro Conference on Real-Time Systems, IEEE, pp 205–214

Real J, Crespo A (2004) Mode change protocols for real-time systems: A survey and a new proposal. Real-time systems 26(2):161–197

Shekhar M, Sarkar A, Ramaprasad H, Mueller F (2012) Semi-partitioned hard-real-time scheduling under locked cache migration in multicore systems. In: Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS), pp 331–340

Sousa PB, Souto P, Tovar E, Bletsas K (2013) The carousel-EDF scheduling algorithm for multiprocessor systems. In: Proceedings of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp 12–21

Voronov S, Anderson JH (2018) An optimal semi-partitioned scheduler assuming arbitrary affinity masks. In: Proceedings of the 39th IEEE Real-Time Systems Symposium (RTSS), pp 408–420

A Additional Figures

In this appendix, we present additional experimental results that were omitted from Sec. 6 for brevity. We first show the full results of our schedulability study from Sec. 6.1, followed by the corresponding plots of minimum schedulable container periods. Finally, we show the mean analytical tardiness bounds that were computed for the dynamic task systems generated in Sec. 6.2.

A.1 Schedulability Experiments

In this section, we provide full results of the schedulability study from Sec. 6.1. The same general trend observed there holds throughout, with EDF-sc giving high weighted schedulability when periods are short, and both schedulers giving high weighted schedulability for task systems with longer periods.

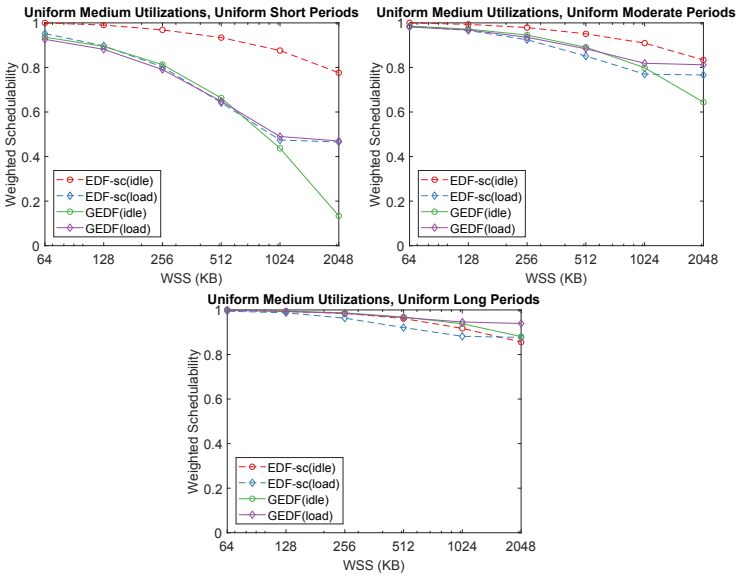


Fig. 13: Schedulability results for medium task sets generated with a uniform distribution.

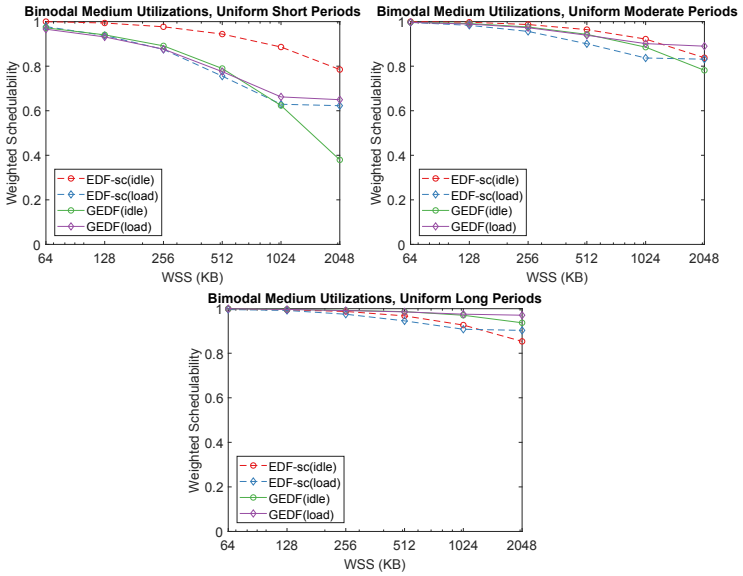


Fig. 14: Schedulability results for medium task sets generated with a bimodal distribution.

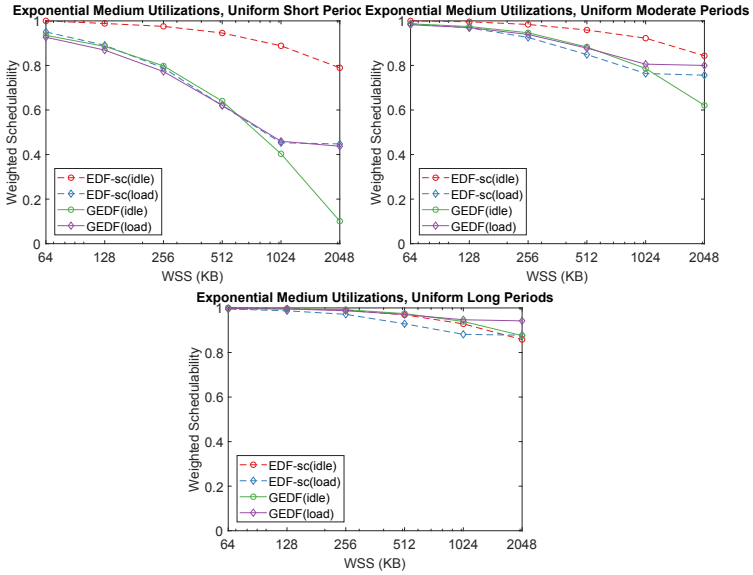


Fig. 15: Schedulability results for medium task sets generated with an exponential distribution.

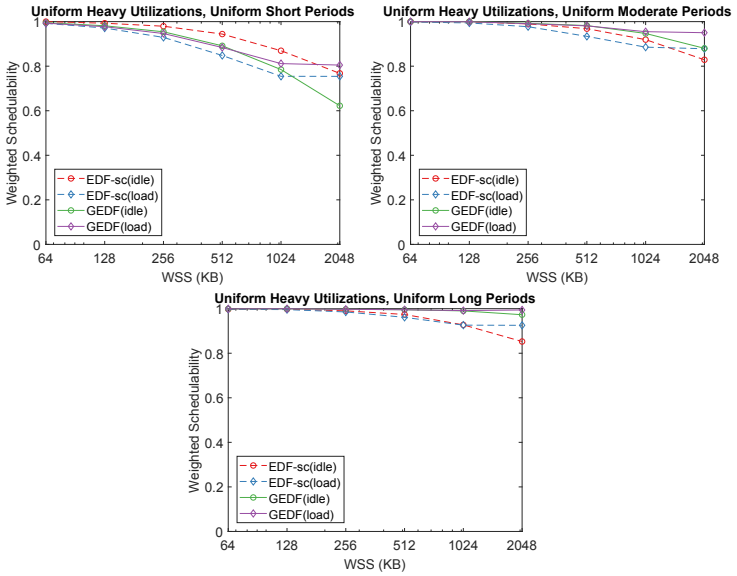


Fig. 16: Schedulability results for heavy task sets generated with a uniform distribution.

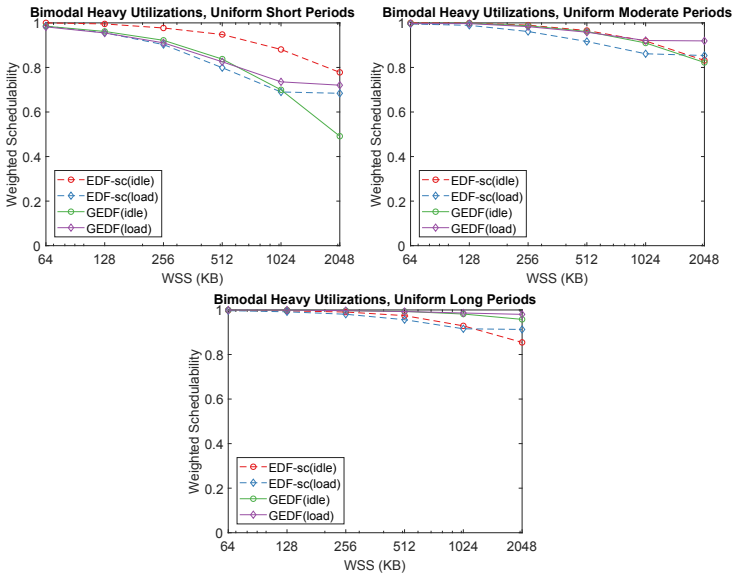


Fig. 17: Schedulability results for heavy task sets generated with a bimodal distribution.

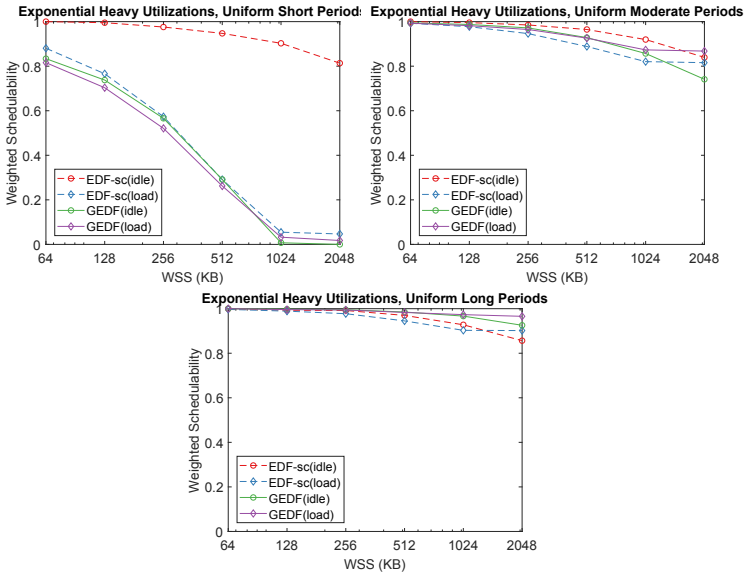


Fig. 18: Schedulability results for heavy task sets generated with an exponential distribution.

A.2 Container Period Experiments

In this section, we show the full set of plots of the minimum container periods for which the task sets from the experiments in Sec. 6.1 were schedulable with EDF-sc. As mentioned in the main text, these curves all show a sharp upward trend once the container period reaches some value. This period at which this occurs varies between the different task set parameters, but is not more than 50 ms in any case, so this was used as a conservative container period in the subsequent experiments.

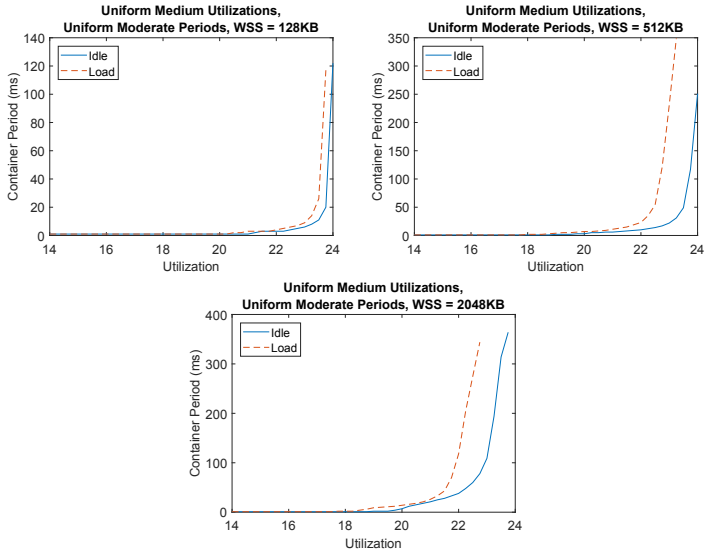


Fig. 19: Minimum container periods required to schedule medium task sets generated with a uniform distribution and medium periods at different WSS.

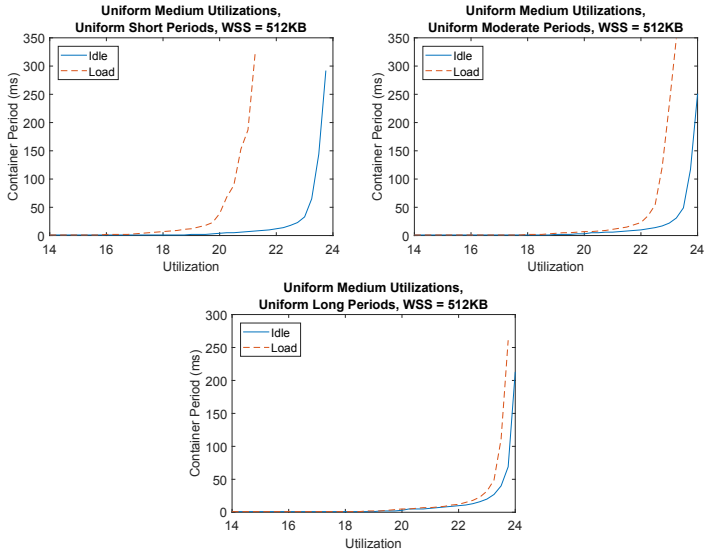


Fig. 20: Minimum container periods required to schedule medium task sets generated with a uniform distribution and differing periods at WSS = 512.

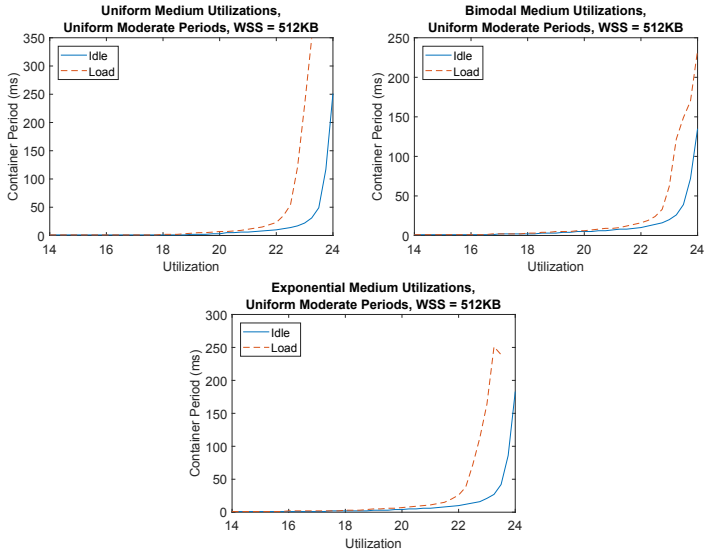


Fig. 21: Minimum container periods required to schedule medium task sets generated with different distributions and medium periods at WSS = 512.

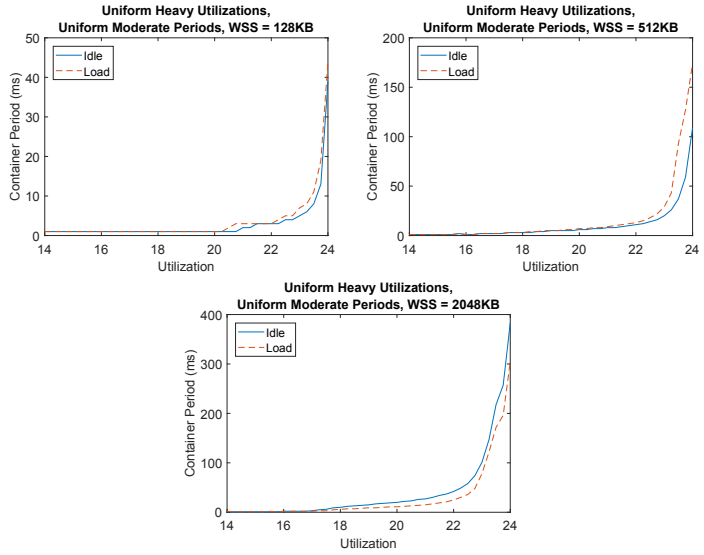


Fig. 22: Minimum container periods required to schedule heavy task sets generated with a uniform distribution and medium periods at different WSS.

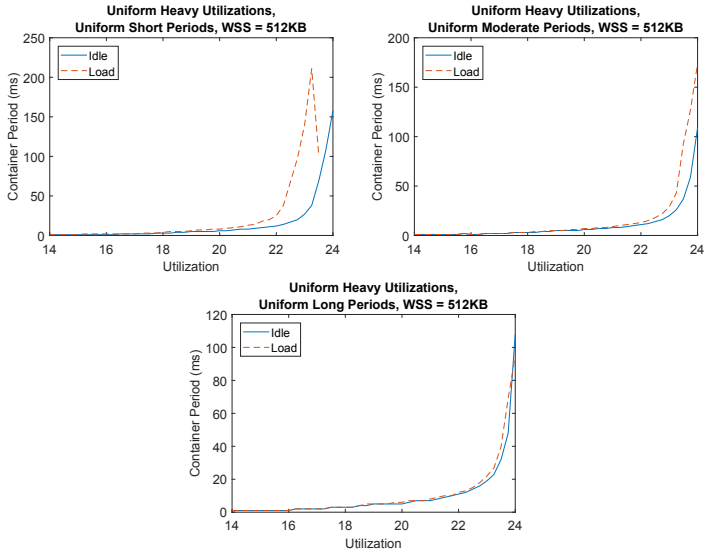


Fig. 23: Minimum container periods required to schedule heavy task sets generated with a uniform distribution and differing periods at WSS $= 512$.

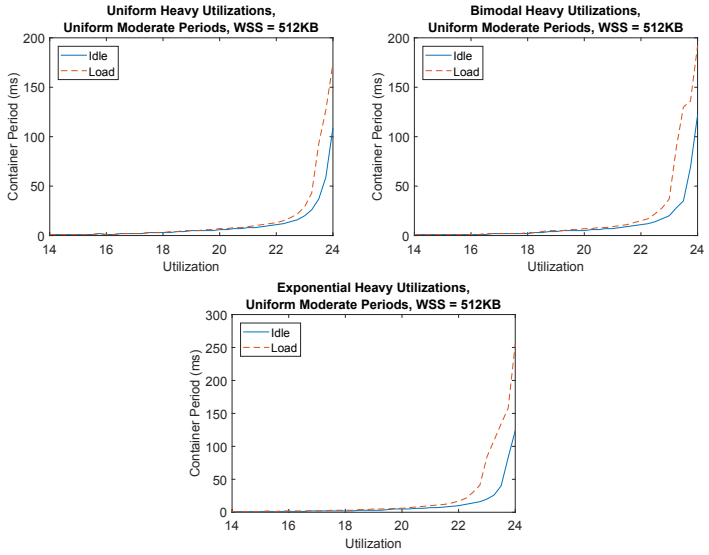


Fig. 24: Minimum container periods required to schedule heavy task sets generated with different distributions and medium periods at WSS $= 512$.

A.3 Tardiness Bounds from Heuristic Comparison

Finally, in this section, we show the mean tardiness bounds for the dynamic task systems generated in Sec. 6.2.

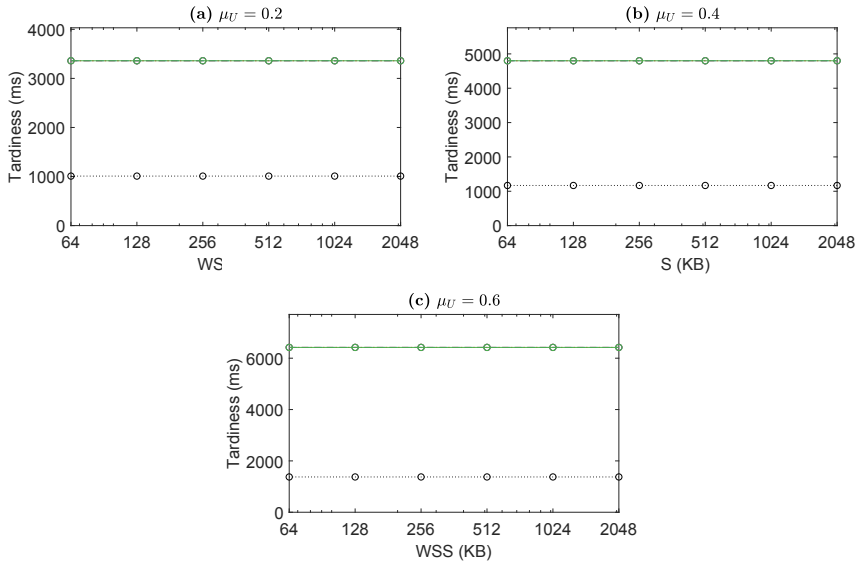


Fig. 25: Mean tardiness bounds for easy- (a), moderate- (b), and hard-to-partition (c) dynamic workloads, as computed using Corollaries 1 and 2. Note the different scales on the Y axis.