Optimal Soft Real-Time Semi-Partitioned Scheduling Made Simple (And Dynamic)

Clara Hobbs
Univ. of North Carolina at Chapel Hill
cghobbs@cs.unc.edu

Zelin Tong
Univ. of North Carolina at Chapel Hill
ztong@cs.unc.edu

James H. Anderson
Univ. of North Carolina at Chapel Hill
anderson@cs.unc.edu

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

ACM Reference Format:
Clara Hobbs, Zelin Tong, and James H. Anderson. 2019. Optimal Soft Real-Time Semi-Partitioned Scheduling Made Simple (And Dynamic). In *27th International Conference on Real-Time Networks and Systems (RTNS 2019), November 6–8, 2019, Toulouse, France.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3356401.3356402

ACKNOWLEDGMENTS

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.

RTNS 2019, November 6–8, 2019, Toulouse, France

ACM ISBN 978-1-4503-7223-7/19/11…$15.00
https://doi.org/10.1145/3356401.3356402

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-fm [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 τ 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 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*[2] 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.

[1]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.

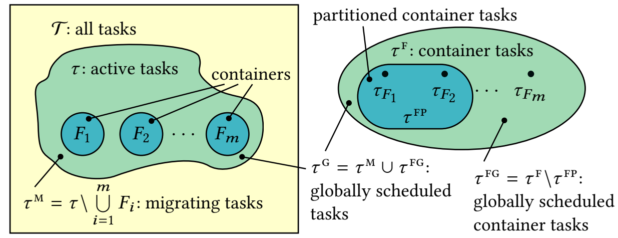

Figure 1: The relationships between the sets used in EDF-sc.

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 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 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

[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].

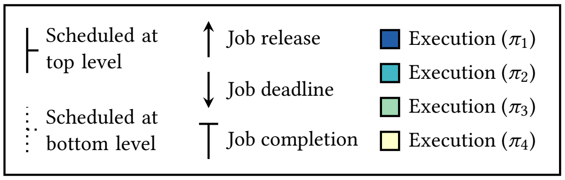

Figure 2: Legend for Figs. 3, 4, 5, and 6.

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.

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

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

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.

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 $|\pi^{\mathrm{G}}|$ 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

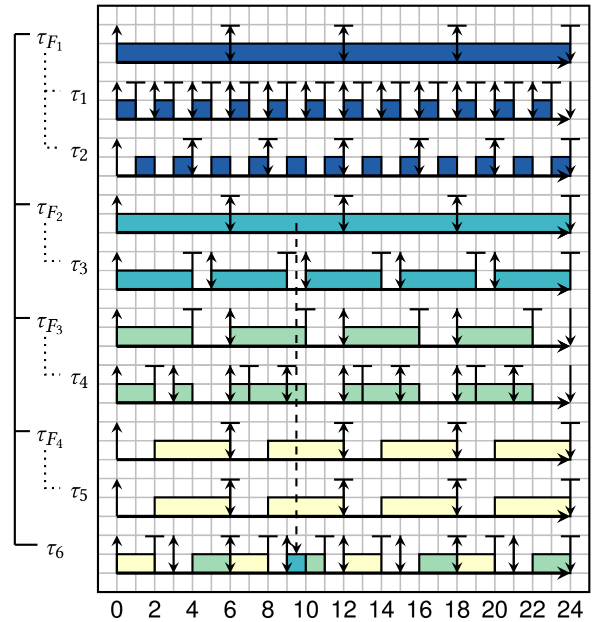

Figure 3: The schedule described in Example 3.1.

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 $\tau_1 = (1, 2)$, $\tau_2 = (2, 4)$, $\tau_3 = (4, 5)$, $\tau_4 = (2, 3)$, $\tau_5 = (4, 6)$, 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 [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 τ, as well as changing container weights.[3] 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 [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 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.

[3]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.

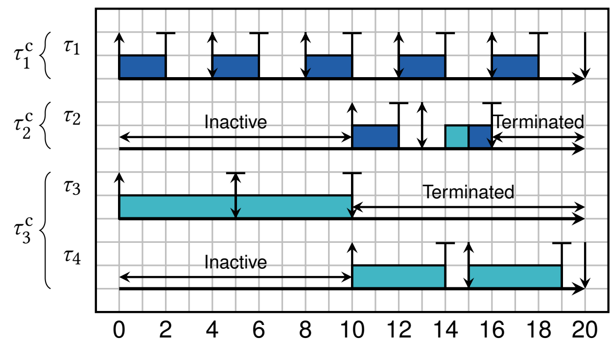

Figure 4: The extended sporadic task system in Example 4.1.

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* $\tau_1^c, \tau_2^c, \ldots, \tau_n^c$ 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[4]

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

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 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 $\tau_1^c = \{\tau_1 : (2,4)\}$, $\tau_2^c = \{\tau_2 : (2,3)\}$, and $\tau_3^c = \{\tau_3 : (5,5), \tau_4 : (4,5)\}$. 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 4.2. *Migrating and container tasks in EDF-sc can be modeled by Devi's extended sporadic task model.*

[4]New tardiness analysis techniques for GEDF [18, 27] 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 is beyond the scope of what we can present in the space allowed.

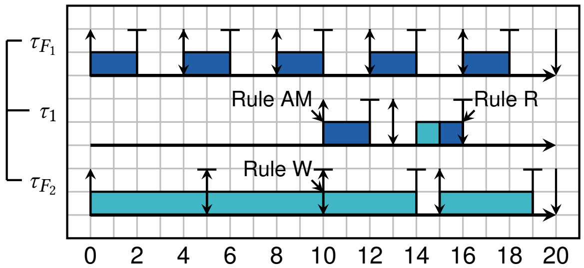

Figure 5: An EDF-sc schedule analogous to the one in Fig. 4.

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 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 4.2 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 4.3. *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.

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 τ^G to be scheduled using GEDF on $|\pi^G| < m$ processors. Thus, by Lemma 4.2, tardiness for a task $\tau_i \in \tau^G$ is upper-bounded by

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

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 $\tau^M \cup \tau^F$ excluded from the summation in the denominator of (9) are the fully provisioned containers τ^{FP}, whose total maximum utilization is $m - |\tau^G|$. Therefore, tardiness for tasks in τ^G 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 π_i is at most*

$$\sigma_i = 2T_{F_i} + \frac{\sum_{\tau_z \in \mathscr{C}^{\max}(\tau^M \cup \tau^F, m-1)} C_z^{\max}}{m - \sum_{\tau_z \in \mathscr{U}^{\max}(\tau^M \cup \tau^F, m-2)} U_z^{\max}} + C_{F_i}^{\max}. \tag{10}$$

Proof sketch. 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 $d_{i,j}$. 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 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 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 $\tau_{a,b}$. 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.

The processor demand created by jobs of fixed tasks on π_i over an interval $[r, s)$ is at most

$$\alpha_i(r, s) = \int_r^s U\big(F_i(t)\big)\ dt.$$

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

Now we derive a lower bound to processor availability to fixed tasks on π_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 Δ of the form

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

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 [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 $[r, s)$ as described above.

By examining the scheduling of τ_{F_i}, 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 to the tardiness bound from Theorem 4.3, 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 argue that the processor availability over any interval $[r, s)$ is at least

$$\beta_i(r, s) = \max\left(0, \int_r^{s-\sigma_i} U\big(F_i(t)\big)\ dt\right).$$

The reweighting rules in Sec. 3.2 guarantee that at any time t, $U(F_i(t)) \le U_{F_i}(t)$, 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 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, $\alpha_i(t_0, d_{a,b})$, equals the minimum processor supply from t_0 to $d_{a,b} + \sigma_i$, $\beta_i(t_0, d_{a,b} + \sigma_i)$. 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. □

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 [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 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* [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 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 $F'_1, \dots, F'_m, \tau^{M\prime}$. This heuristic uses a bin-packing heuristic to find a container F'_i so that $U(F'_i) + U_a \le 1$ holds.

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: $F'_1, \ldots, F'_m, \tau^{\mathrm{M}\prime} \leftarrow F_1, \ldots, F_m, \tau^{\mathrm{M}}$
▷ Determine which tasks can be added
11: **while** PendingAdds is not empty **do**
12: $\tau_a \leftarrow$ PendingAdds.Dequeue()
13: Select container F'_i, $\tau^{\mathrm{M}\prime}$, 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** $\tau_k \in \tau^{\mathrm{M}}$ **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** $\tau_k \in F'_i \backslash F_i$ **do**
25: AF(τ_k, F_i)
26: **for each** $\tau_k \in \tau^{\mathrm{M}\prime} \backslash \tau^{\mathrm{M}}$ **do**
27: AM(τ_k)

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 $\sum_{i=1}^{m} U(F'_i) + U(\tau^{\mathrm{M}\prime}) + U_a \leq m$ 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.

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.

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_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.)

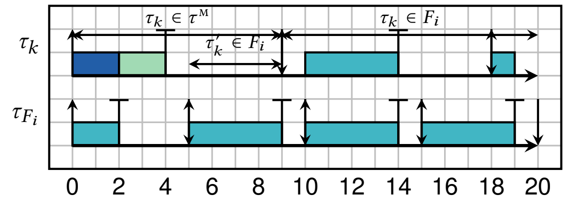

Figure 6: Task τ_k is eligible to be moved at time 5, since $\tau_{k,1}$ completed at time 4, and $d_{k,1} = 9 < d_{F_i,2} = 10$. Its copy τ'_k is added to F_i at time 5, and at time 9 the move is completed. Such "copy" tasks merely reserve container capacity and are not executed.

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, 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.

Container task reweighting. Once all operations that add tasks to containers have been fully planned, we call a ContainerReweightingHeuristic at line 21 to determine new container weights based on the sets $F'_1, \ldots, F'_m, \tau^{\mathrm{M}\prime}$. 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 $\Sigma_{i=1}^{m} U_{F_i} + U(\tau^{\mathrm{M}\prime}) \leq m$ 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 $U(\tau^{\mathrm{G}}) \leq |\pi^{\mathrm{G}}|$, 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. 7b)** Fully provision container tasks as with MinOrFull, but over-provision the others, each by an equal fraction of the extra processor capacity $|\pi^{\mathrm{G}}| - U(\tau^{\mathrm{G}})$.

Once the new weights have been determined by one of these heuristics, the container tasks are all reweighted using Rule W at line 22.

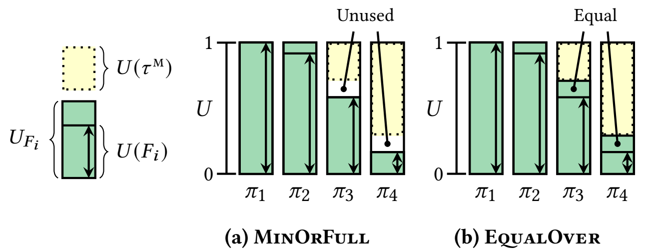

Figure 7: The two container task provisioning techniques discussed. (a) Container tasks τ_{F_3} and τ_{F_4} are minimally provisioned ($U_{F_i} = U(F_i)$), and τ_{F_1} and τ_{F_2} are fully provisioned ($U_{F_i} = 1$). (b) Container task utilizations U_{F_3} and U_{F_4} are increased equally to consume the system capacity left unused by MinOrFull.

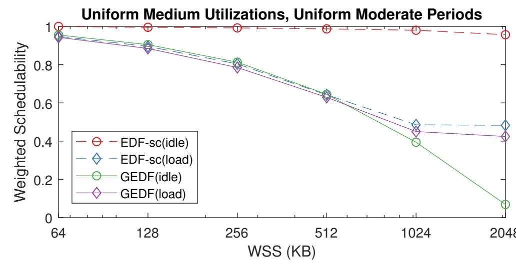

Figure 8: Weighted schedulability for task sets with uniform medium utilizations and uniform short periods.

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

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 large 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.

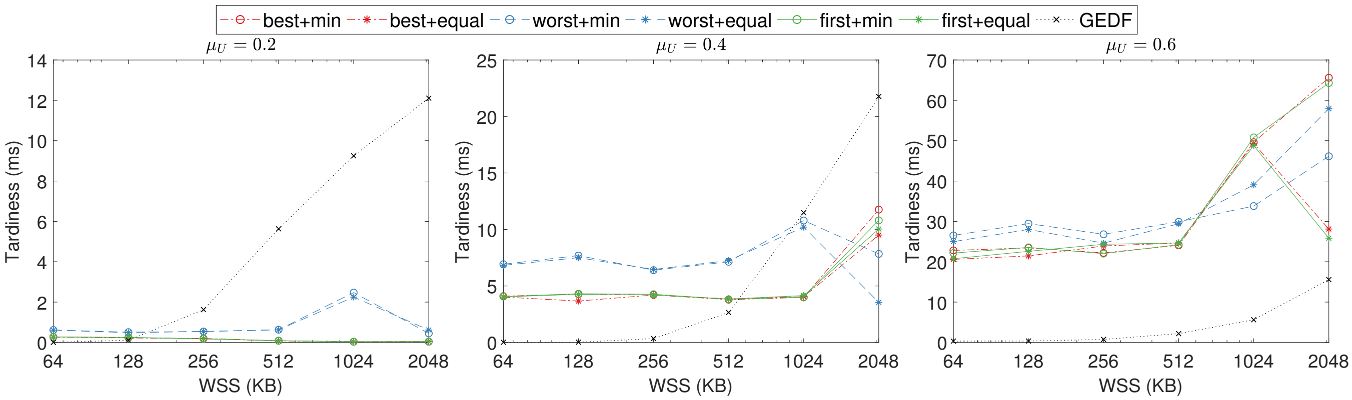

Figure 9: Left to right: observed average tardiness over easy-, moderate-, and hard-to-partition workloads.

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 $\mathcal{T}$, 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 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 $\Lambda = (1-U(\tau)/m)+\psi(U(\tau)/m)$ 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 $\mathcal{T}\backslash\tau$ 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 [14]. 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 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

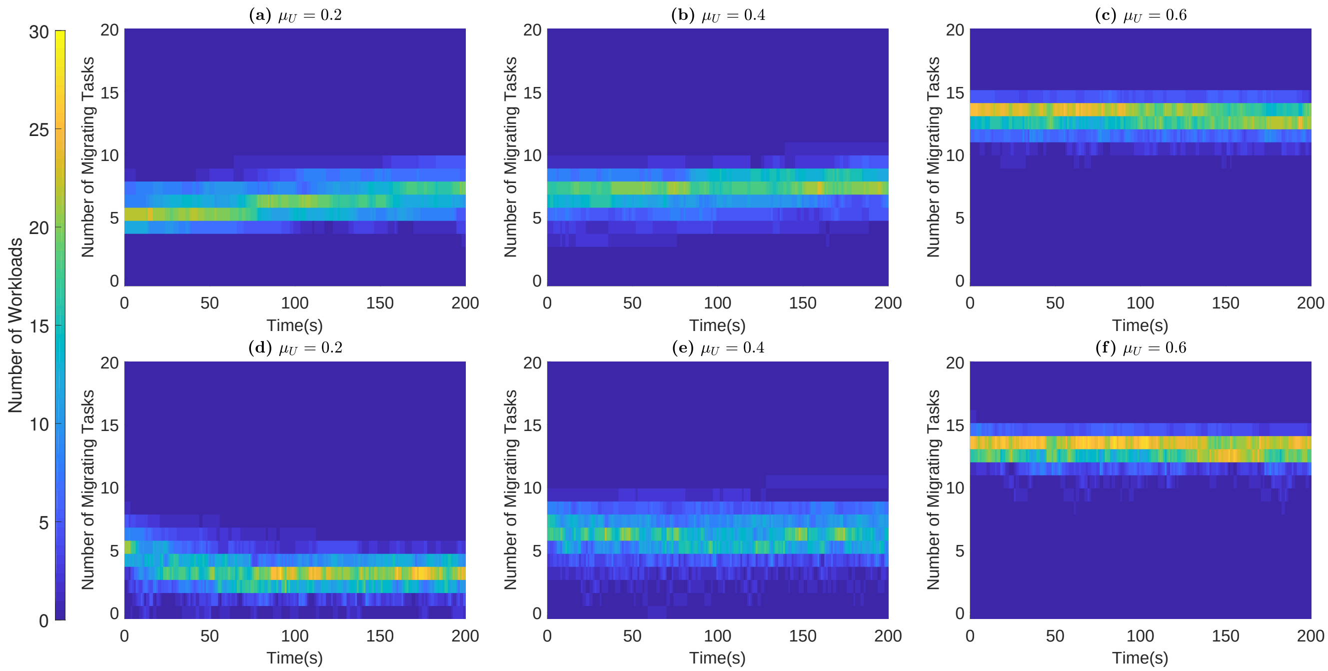

Figure 10: Stabilization heatmaps for WSS of 2048 kB. Insets (a), (b), and (c) (top) show easy-, moderate-, and hard-to-partition workloads with stabilization disabled. Insets (d), (e), and (f) (bottom) show easy-, moderate-, and hard-to-partition workloads with stabilization enabled.

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

[1] James H. Anderson, Vasile Bud, and UmaMaheswari C. Devi. 2005. An EDF-based Scheduling Algorithm for Multiprocessor Soft Real-Time Systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS).*

[2] James H. Anderson, Jeremy P. Erickson, UmaMaheswari C. Devi, and Benjamin N. Casses. 2016. Optimal semi-partitioned scheduling in soft real-time systems. *Journal of Signal Processing Systems* 84, 1 (2016), 3–23.

[3] Björn Andersson, Konstantinos Bletsas, and Sanjoy K. Baruah. 2008. Scheduling arbitrary-deadline sporadic task systems on multiprocessors. In *Proceedings of the 29th IEEE Real-Time Systems Symposium (RTSS).* 385–394.

[4] Björn Andersson and Eduardo Tovar. 2006. Multiprocessor scheduling with few preemptions. In *Proceedings 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA).* 322–334.

[5] Andrea Bastoni, Björn B. Brandenburg, and James H. Anderson. 2010. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. *Proceedings of OSPERT* (2010), 33–44.

[6] Andrea Bastoni, Björn B. Brandenburg, and James H. Anderson. 2011. Is semi-partitioned scheduling practical?. In *2011 23rd Euromicro Conference on Real-Time Systems.* IEEE, 125–135.

[7] M. Khurram Bhatti, Cécile Belleudy, and Michel Auguin. 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).* 1594–1601.

[8] Konstantinos Bletsas and Björn Andersson. 2009. Notional processors: an approach for multiprocessor scheduling. In *proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).* 3–12.

[9] Konstantinos Bletsas and Björn Andersson. 2011. Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. *Real-Time Systems* 47, 4 (2011), 319–355.

[10] Aaron Block, James H. Anderson, and Gary Bishop. 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).* 429–435.

[11] Aaron Block, James H. Anderson, and UmaMaheswari C. Devi. 2008. Task reweighting under global scheduling on multiprocessors. *Real-Time Systems* 39, 1-3 (2008), 123–167.

[12] Björn B. Brandenburg and Mahircan Gül. 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).* 99–110.

[13] Alan Burns, Robert I. Davis, Pengyu Wang, and Fengxiang Zhang. 2012. Partitioned EDF scheduling for multiprocessors using a C = D task splitting scheme. *Real-Time Systems* 48, 1 (2012), 3–33.

[14] Daniel Casini, Alessandro Biondi, and Giorgio Buttazzo. 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).* 13:1–13:23.

[15] UmaMaheswari C. Devi. 2006. *Soft Real-Time Scheduling on Multiprocessors.* Ph.D. Dissertation. University of North Carolina at Chapel Hill.

[16] UmaMaheswari C. Devi and James H. Anderson. 2008. Tardiness bounds under global EDF scheduling on a multiprocessor. *Real-Time Systems* 38, 2 (2008), 133–189.

[17] François Dorin, Patrick Meumeu Yomsi, Joël Goossens, and Pascal Richard. 2010. Semi-Partitioned Hard Real-Time Scheduling with Restricted Migrations upon Identical Multiprocessor Platforms. *CoRR* abs/1006.2637 (2010). arXiv:1006.2637 http://arxiv.org/abs/1006.2637

[18] Jeremy P. Erickson, UmaMaheswari C. Devi, and Sanjoy K. Baruah. 2010. Improved Tardiness Bounds for Global EDF. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS).* 14–23.

[19] Ming Fan and Gang Quan. 2012. Harmonic semi-partitioned scheduling for fixed-priority real-time tasks on multi-core platform. In *Proceedings of the 2012 Design, Automation Test in Europe Conference Exhibition (DATE).* 503–508.

[20] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. 2010. Fixed-priority multiprocessor scheduling: Beyond Liu & Layland utilization bound. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS) Work-in-Progress Session.* 1594–1601.

[21] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. 2010. 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).* 165–174.

[22] Clara Hobbs, Zelin Tong, and James H. Anderson. 2019. Optimal Soft Real-Time Semi-Partitioned Scheduling Made Simple (And Dynamic), full version. http://cs.unc.edu/~anderson/papers.html

[23] Zoran Ivkovic and Errol L. Lloyd. 1998. Fully Dynamic Algorithms for Bin Packing: Being (Mostly) Myopic Helps. *SIAM J. Comput.* 28, 2 (1998), 574–38.

[24] Shinpei Kato and Nobuyuki Yamasaki. 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).* 441–450.

[25] Shinpei Kato and Nobuyuki Yamasaki. 2008. Portioned EDF-based Scheduling on Multiprocessors. In *Proceedings of the 8th ACM International Conference on Embedded Software (EMSOFT).* 139–148.

[26] Shinpei Kato and Nobuyuki Yamasaki. 2009. Semi-partitioned Fixed-Priority Scheduling on Multiprocessors. In *Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).* 23–32.

[27] Mauro Leoncini, Manuela Montangero, and Paolo Valente. 2018. A parallel branch-and-bound algorithm to compute a tighter tardiness bound for preemptive global EDF. *Real-Time Systems* (2018), 1–38. https://doi.org/10.1007/s11241-018-9319-6

[28] Hennadiy Leontyev, Samarjit Chakraborty, and James H. Anderson. 2011. Multiprocessor extensions to real-time calculus. *Real-Time Systems* 47, 6 (2011), 562.

[29] Aloysius K Mok, Xiang Feng, and Deji Chen. 2001. Resource partition for real-time systems. In *Proceedings of the Seventh IEEE Real-Time Technology and Applications Symposium (RTAS).* 75–84.

[30] Mayank Shekhar, Abhik Sarkar, Harini Ramaprasad, and Frank Mueller. 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).* 331–340.

[31] Paulo B. Sousa, Pedro Souto, Eduardo Tovar, and Konstantinos Bletsas. 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).* 12–21.

[32] Sergey Voronov and James H. Anderson. 2018. An Optimal Semi-Partitioned Scheduler Assuming Arbitrary Affinity Masks. In *Proceedings of the 39th IEEE Real-Time Systems Symposium (RTSS).* 408–420.

A FIGURES

A.1 Schedulability Experiments

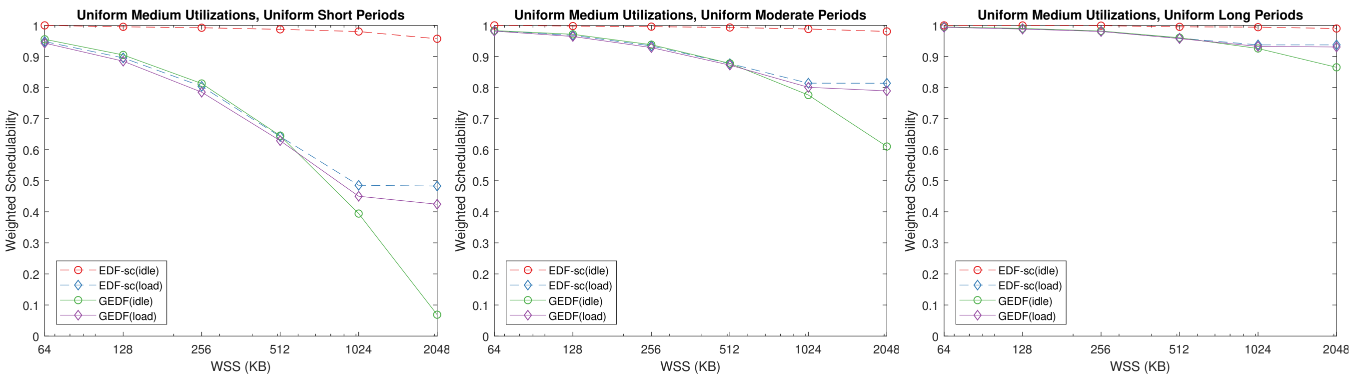

Figure 11: Schedulability results for medium task sets generated with a uniform distribution.

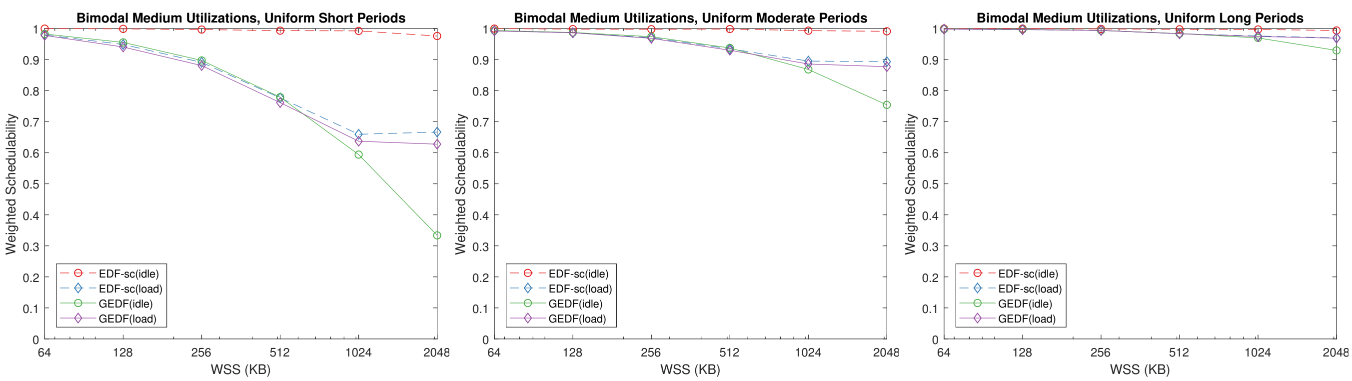

Figure 12: Schedulability results for medium task sets generated with a bimodal distribution.

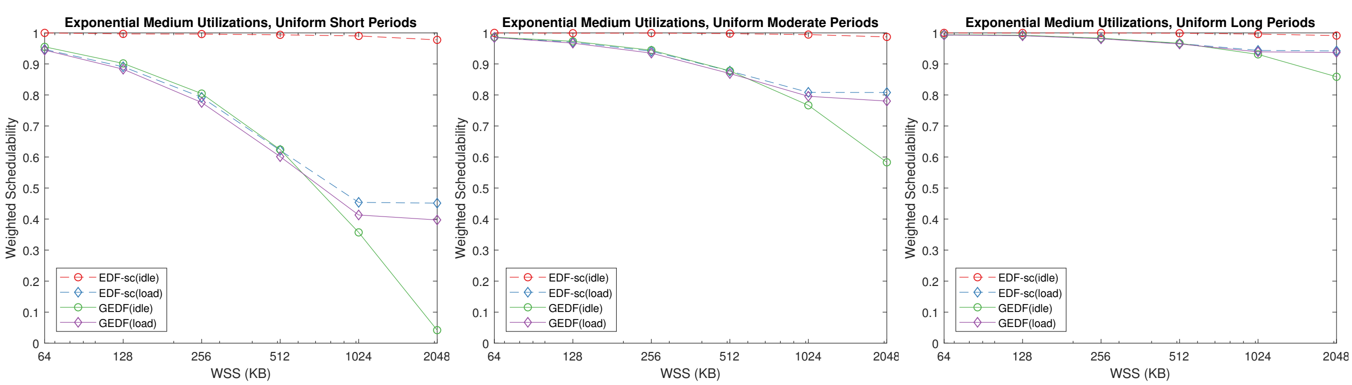

Figure 13: Schedulability results for medium task sets generated with an exponential distribution.

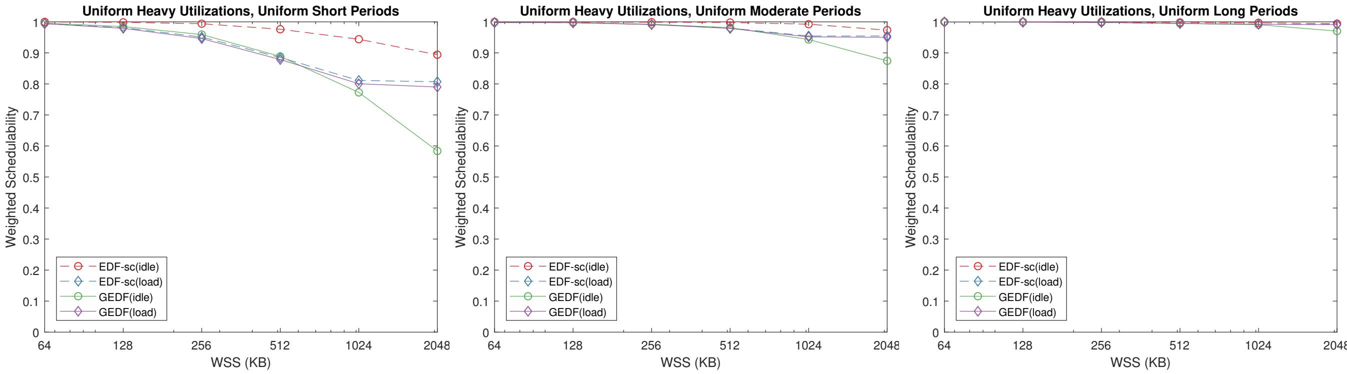

Figure 14: Schedulability results for heavy task sets generated with a uniform distribution.

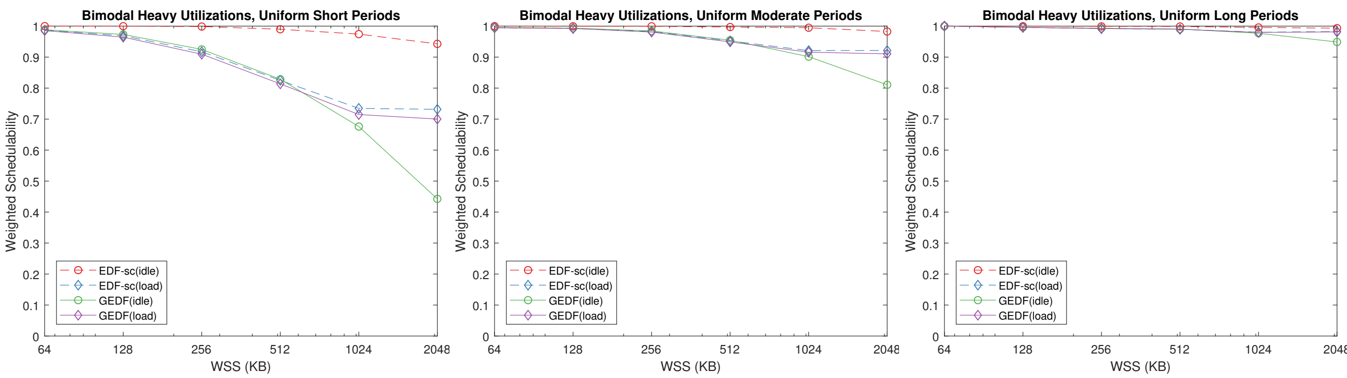

Figure 15: Schedulability results for heavy task sets generated with a bimodal distribution.

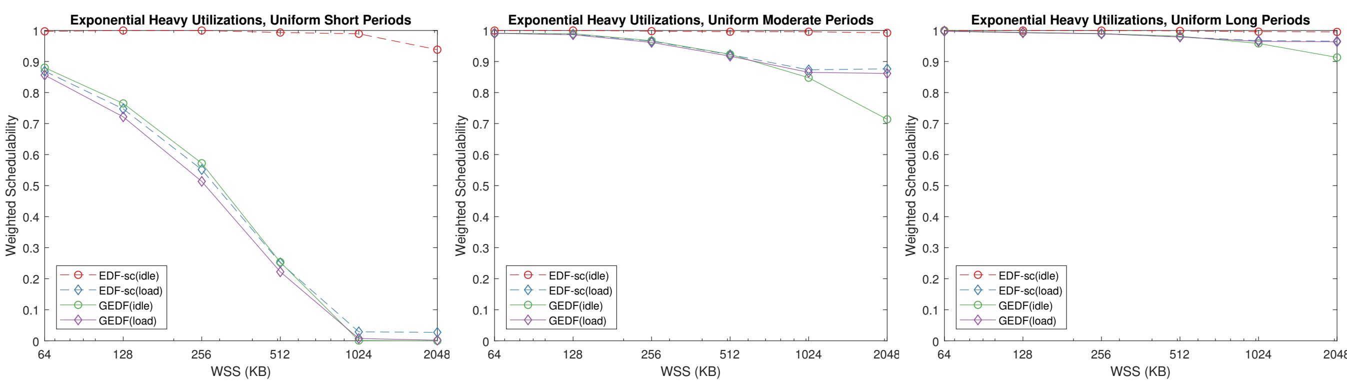

Figure 16: Schedulability results for heavy task sets generated with an exponential distribution.

A.2 Container Period Experiments

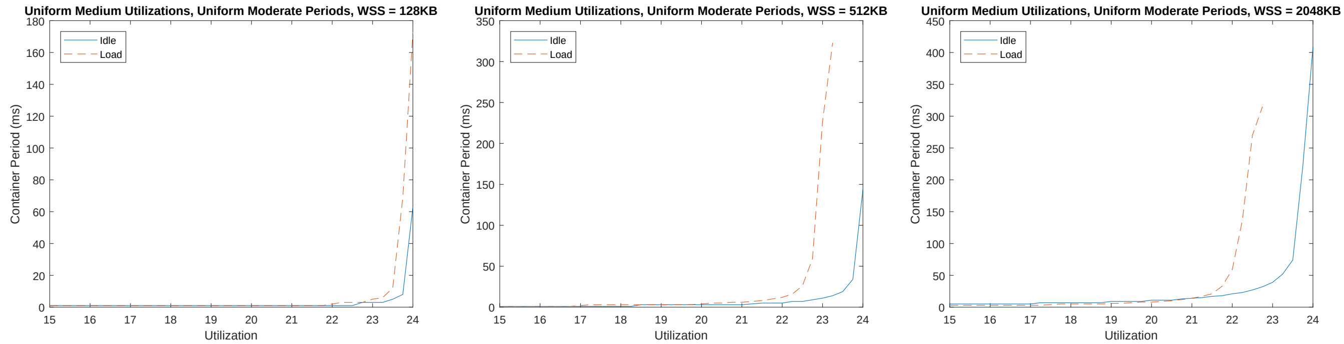

Figure 17: Minimum container periods required to schedule medium task sets generated with a uniform distribution and medium periods at different WSS.

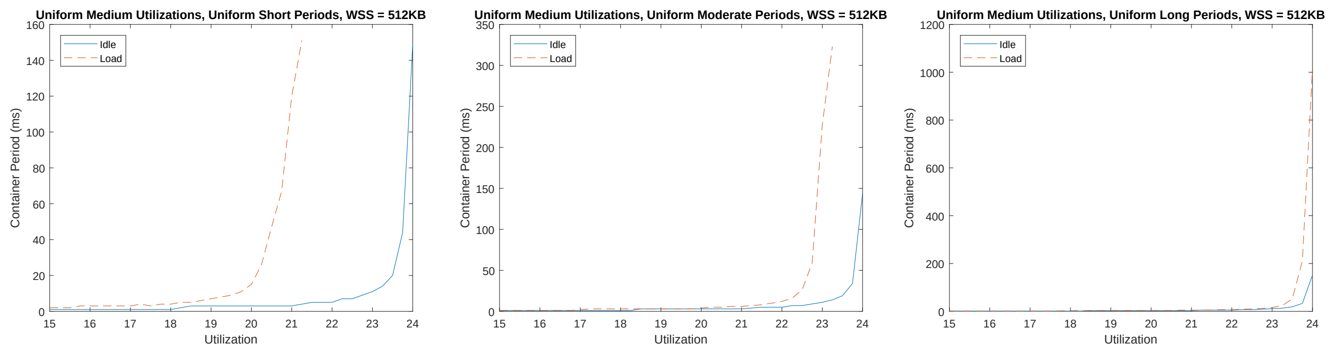

Figure 18: Minimum container periods required to schedule medium task sets generated with a uniform distribution and differing periods at WSS = 512.

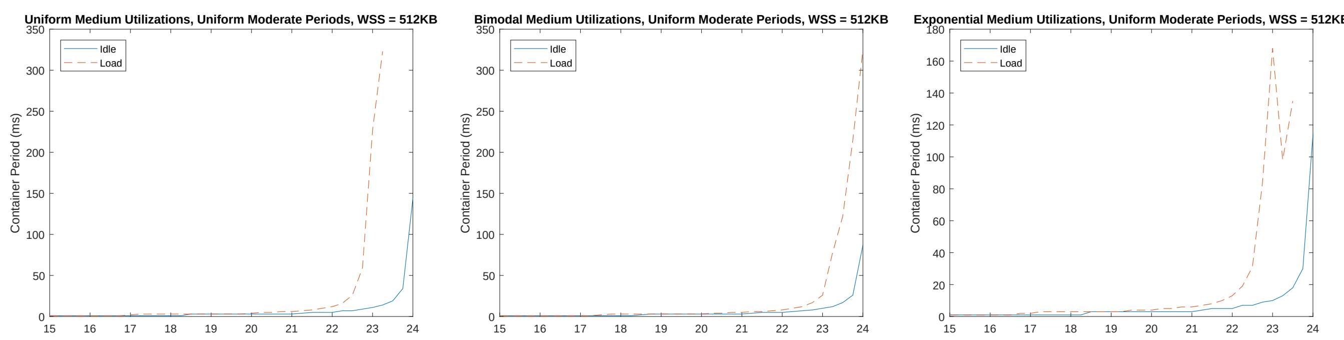

Figure 19: Minimum container periods required to schedule medium task sets generated with different distributions and medium periods at WSS = 512.

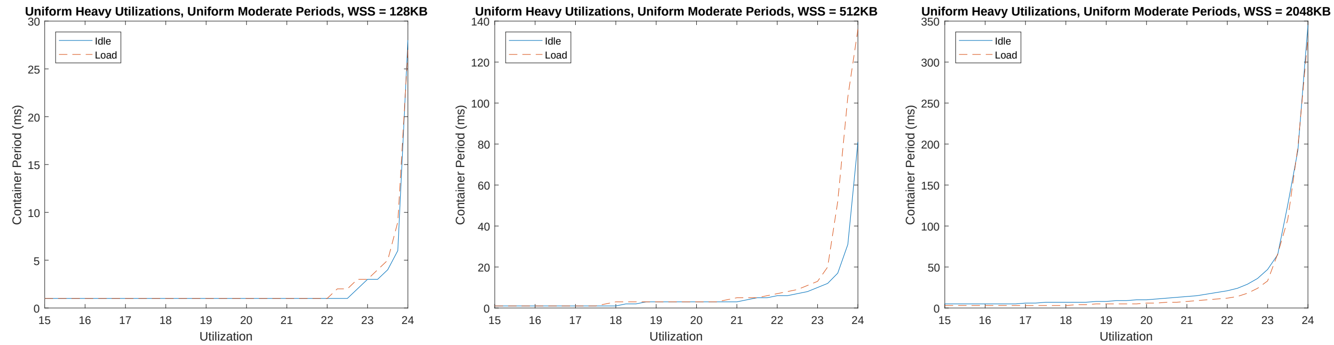

Figure 20: Minimum container periods required to schedule heavy task sets generated with a uniform distribution and medium periods at different WSS.

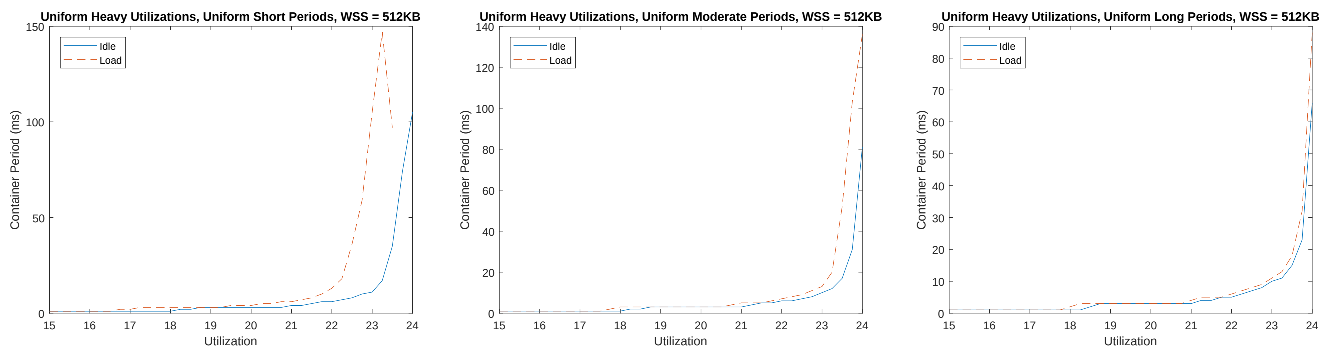

Figure 21: Minimum container periods required to schedule heavy task sets generated with a uniform distribution and differing periods at WSS = 512.

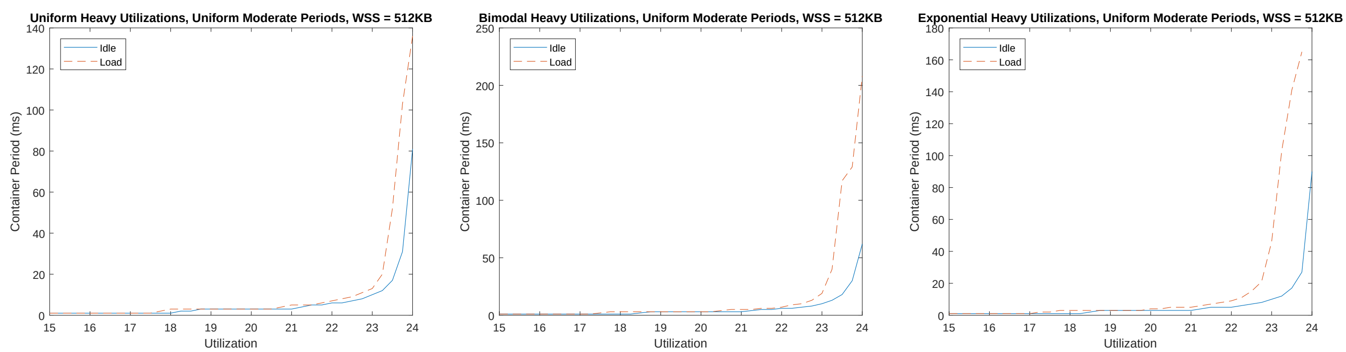

Figure 22: Minimum container periods required to schedule heavy task sets generated with different distributions and medium periods at WSS = 512.

A.3 Tardiness Experiments

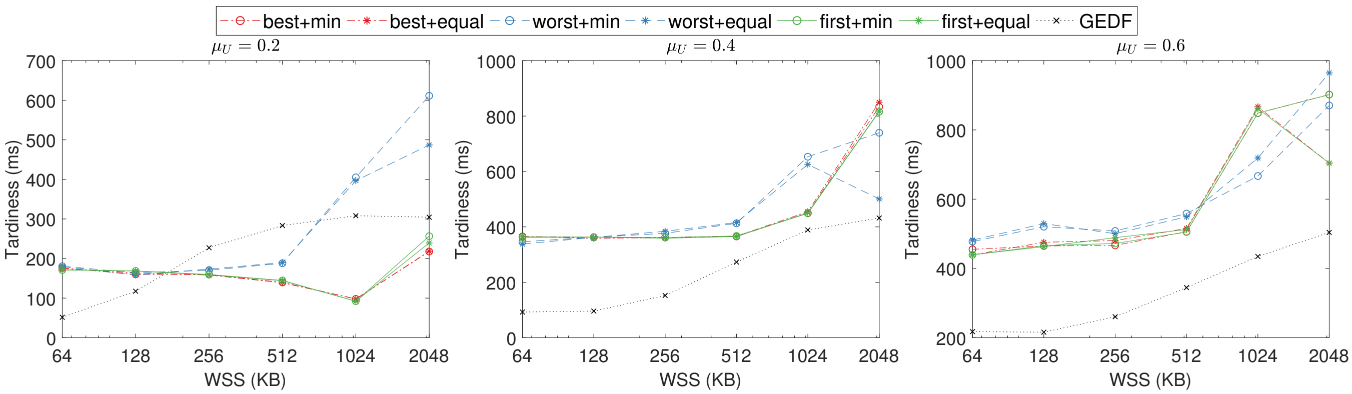

Figure 23: Left to right: observed maximum tardiness over easy-, moderate-, and hard-to-partition workloads.

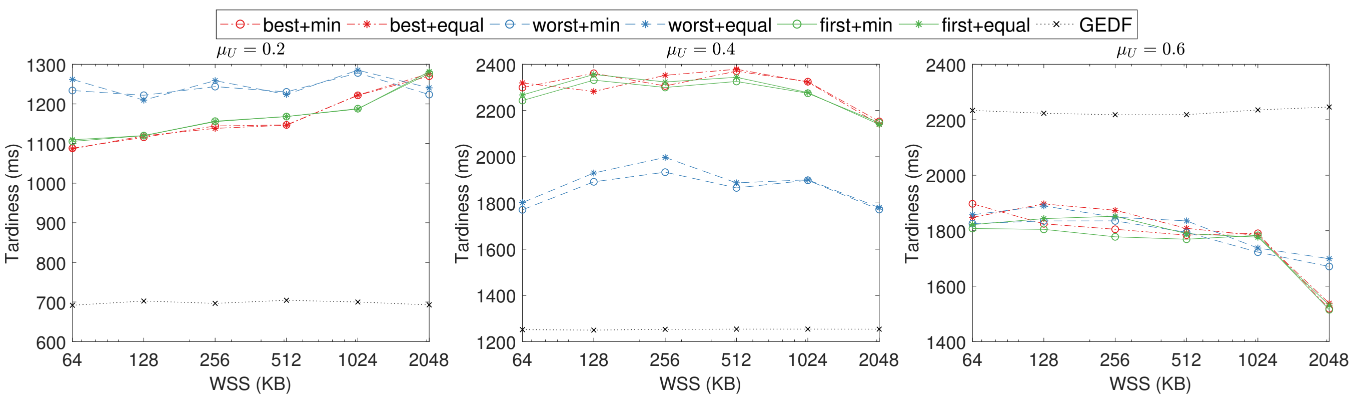

Figure 24: Left to right: tardiness bound over easy-, moderate-, and hard-to-partition workloads.

A.4 Migrating Task Experiments

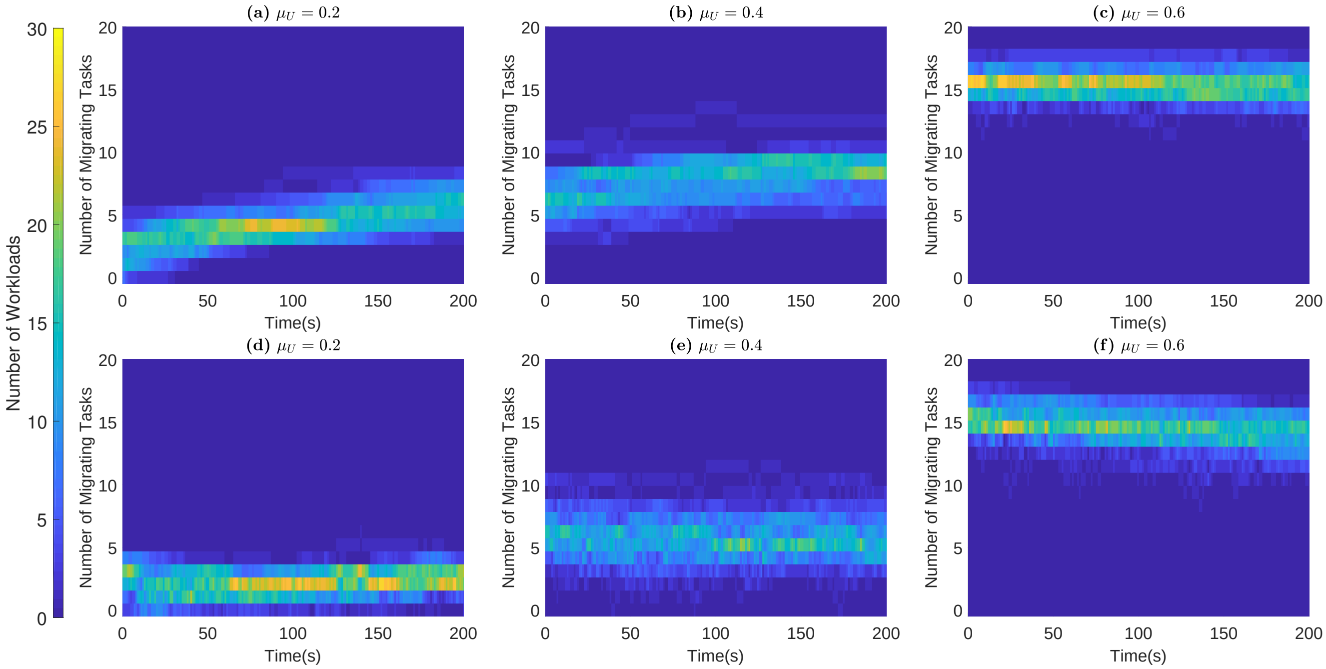

Figure 25: Stabilization heatmaps for WSS of 64 kB. Insets (a), (b), and (c) (top) show easy-, moderate-, and hard-to-partition workloads with stabilization disabled. Insets (d), (e), and (f) (bottom) show easy-, moderate-, and hard-to-partition workloads with stabilization enabled.

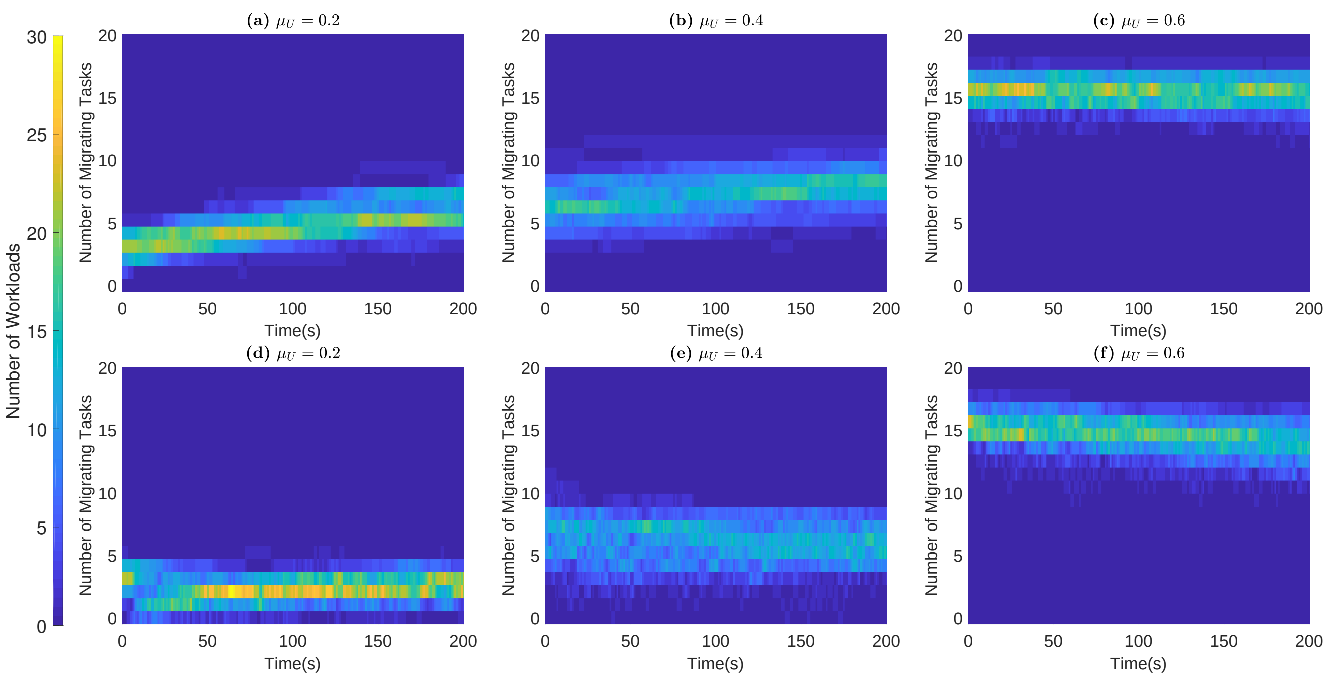

Figure 26: Stabilization heatmaps for WSS of 128 kB. Insets (a), (b), and (c) (top) show easy-, moderate-, and hard-to-partition workloads with stabilization disabled. Insets (d), (e), and (f) (bottom) show easy-, moderate-, and hard-to-partition workloads with stabilization enabled.

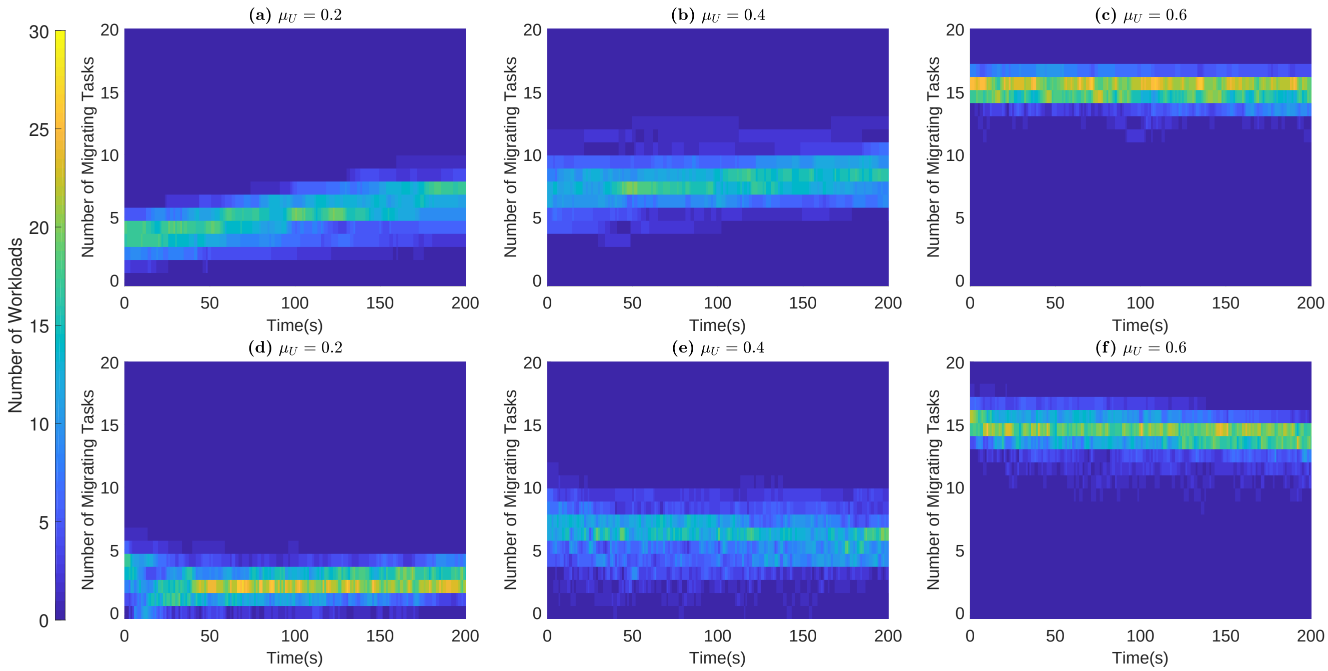

Figure 27: Stabilization heatmaps for WSS of 256 kB. Insets (a), (b), and (c) (top) show easy-, moderate-, and hard-to-partition workloads with stabilization disabled. Insets (d), (e), and (f) (bottom) show easy-, moderate-, and hard-to-partition workloads with stabilization enabled.

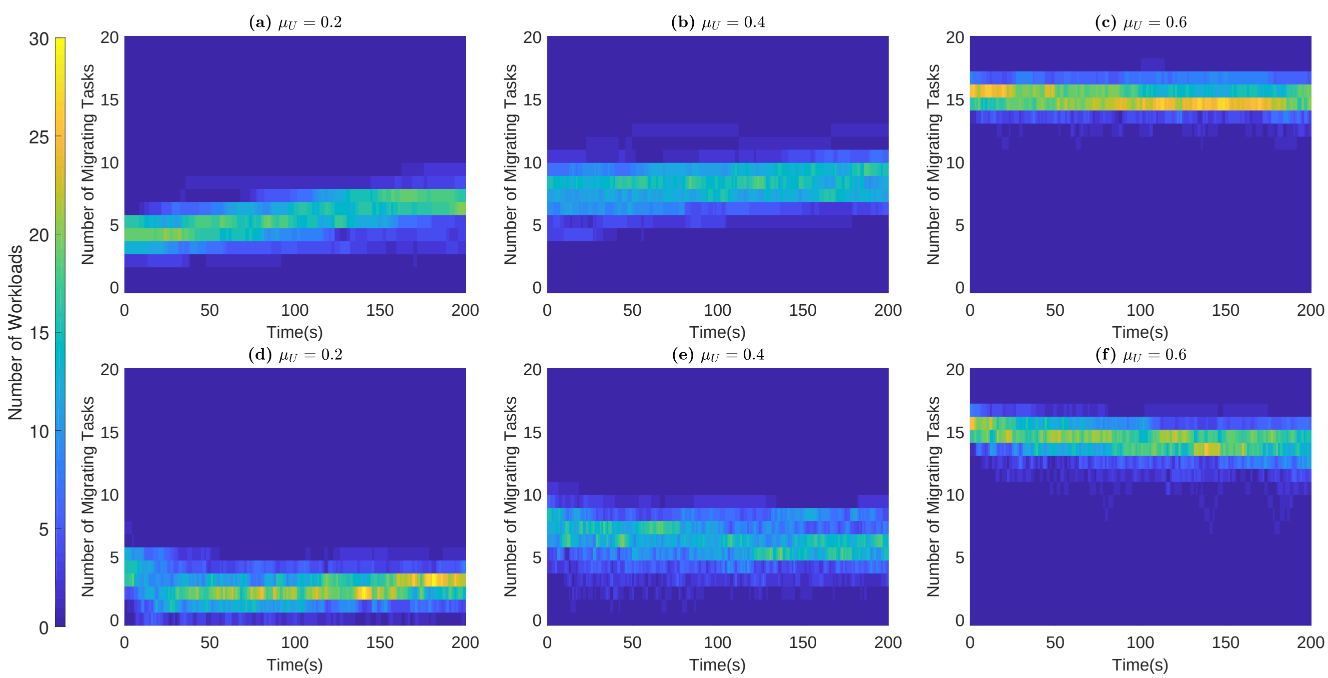

Figure 28: Stabilization heatmaps for WSS of 512 kB. Insets (a), (b), and (c) (top) show easy-, moderate-, and hard-to-partition workloads with stabilization disabled. Insets (d), (e), and (f) (bottom) show easy-, moderate-, and hard-to-partition workloads with stabilization enabled.

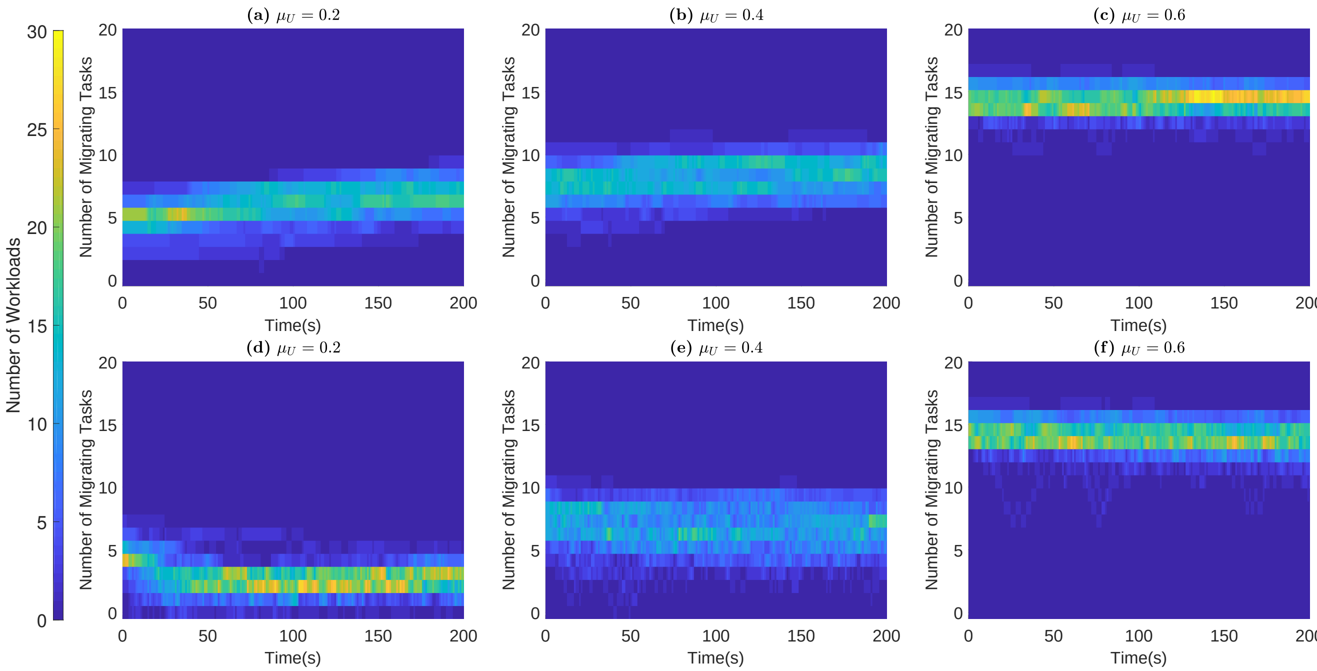

Figure 29: Stabilization heatmaps for WSS of 1024 kB. Insets (a), (b), and (c) (top) show easy-, moderate-, and hard-to-partition workloads with stabilization disabled. Insets (d), (e), and (f) (bottom) show easy-, moderate-, and hard-to-partition workloads with stabilization enabled.