On the Defectiveness of SCHED_DEADLINE w.r.t. Tardiness and Affinities, and a Partial Fix

Stephen Tang  
James H. Anderson  
{sytang, anderson}@cs.unc.edu  
University of North Carolina at Chapel Hill  
Chapel Hill, USA

Luca Abeni  
luca.abeni@santannapisa.it  
Scuola Superiore Sant’Anna  
Pisa, Italy

ABSTRACT

SCHED_DEADLINE (DL for short) is an Earliest-Deadline-First (EDF) scheduler included in the Linux kernel. A question motivated by DL is how EDF should be implemented in the presence of CPU affinities to maintain optimal bounded tardiness guarantees. Recent works have shown that under arbitrary affinities, DL does not maintain such guarantees. Such works have also shown that repairing DL to maintain these guarantees would likely require an impractical overhaul of the existing code. In this work, we show that for the special case where affinities are semi-partitioned, DL can be modified to maintain tardiness guarantees with minor changes. We also draw attention to the fact that admission control is already broken in several respects in the existing DL implementation.

CCS CONCEPTS

• Computer systems organization → Real-time systems.

KEYWORDS

real-time, affinities

ACM Reference Format:


1 INTRODUCTION

SCHED_DEADLINE (DL for short) is an Earliest-Deadline-First (EDF) implementation included in the mainline Linux kernel since version 3.14. Its inclusion has been significant because it has lowered the barrier to entry for real-time EDF scheduling and it has inspired many publications [4, 8, 9, 11, 13, 15, 16].

Admission control. One feature of DL is its admission-control (AC) system. AC prevents the system from being over-utilized by tasks managed by DL. If the total CPU bandwidth required by DL tasks is near some threshold, then AC will reject any requests to create new DL tasks until existing DL tasks reduce their bandwidth consumption. Note that preventing over-utilization is only a necessary condition for guaranteeing that tasks meet deadlines.

Instead of guaranteeing that deadlines will be met, AC satisfies two goals [3]. The first goal is to provide performance guarantees. Specifically, AC guarantees to each DL task that its long-run consumption of CPU bandwidth remains within a bounded margin of error from a requested rate. The second goal is to avoid the starvation of lower-priority non-DL workloads. Specifically, AC guarantees that some user-specified amount of CPU bandwidth will remain for the execution of non-DL workloads. This goal of AC has been an important aspect of Linux real-time scheduling for years, and predates the merging of DL into the kernel.

The theoretical basis of AC is that global EDF scheduling of sporadic real-time tasks on identical multiprocessors guarantees bounded tardiness to jobs if the system is not over-utilized [6]. The DL documentation [2] cites [6]. Bounded tardiness ensures AC’s first goal as it implies that any execution of a job occurs in a finite window around its release and deadline, meaning a task’s rate of execution stays consistent with its bandwidth. Bounded tardiness also prevents any set of DL tasks from having an unbounded number of tardy jobs which, if allowed to execute uninterrupted, could starve non-DL workloads for an unbounded amount of time. Thus, bounded tardiness also supports AC’s second goal.

Affinities. Specifying tasks’ affinities, subsets of the CPUs on which given tasks are permitted to execute, is useful for maintaining cache locality, as tasks whose execution times depend heavily on whether they are cache hot or cold at a certain cache level may benefit from having their affinities restricted to CPUs that share cache at said level. Affinities can also be useful in reducing scheduling-decision overheads, as a CPU need only consider tasks with affinity for that CPU when deciding what to execute.

Unfortunately, the tardiness result in [6] heavily relies on the fact that scheduling is global, and does not hold when tasks have affinities. Thus, with the exception of clustered scheduling (in which each cluster resembles its own global subsystem), the setting of affinities is forbidden in DL unless AC is explicitly disabled.

It was shown in [15] that with arbitrary affinities, under-utilized systems (whose definition becomes more complex under arbitrary affinities) may have unbounded tardiness under DL. Though an EDF variant called SAPA-EDF presented in [5] was proven in [16] to guarantee bounded tardiness with arbitrary affinities for identical CPUs, this variant was not implemented by the DL maintainers due to the complexity of the required modifications to both the scheduler and AC (as the meaning of over-utilization is changed).
This variant would likely have also required higher overheads and increased task migrations in practice.

While [15] showed that the current DL implementation cannot support arbitrary affinities, that work suggested that this may not be the case for special cases of affinities. In this paper, we consider the special case of semi-partitioned (SP) systems, wherein each task has affinity for either one CPU (the task is partitioned) or for all CPUs in its cluster (the task is migrating). While not as flexible as arbitrary affinities, SP affinities still reduce overheads by limiting the number of migrating tasks and can still be used to maintain cache locality for partitioned tasks.

**In practice.** In practice, AC does not guarantee bounded tardiness under DL, even under an idealized abstraction of the scheduler. This is because affinities are far from the the only way that DL differs from the system model considered in [6], from which the soundness of AC originates. In particular, DL has grown to consider dynamic voltage and frequency scaling (DVFS) and asymmetric CPU capacities, both of which break the assumption of identical CPUs in [6]. Besides these changes in the considered platform model, tasks can also be much more dynamic under DL than in [6].

Unlike affinities, AC need not be disabled when using such features. This calls into question the role of AC, as there is no theoretical basis for its goals without tardiness guarantees. Instead of by analysis, these features have often been validated empirically by demonstrating that deadline-miss frequencies are acceptable. As we demonstrate herein, empirical validation is insufficient for showing that AC is not broken.

**Contributions.** We present a DL variant that supports AC for SP systems. This contribution is divided into three parts.

First, we list features supported by DL that were not considered in the analysis in [6]. For a subset of these features, we show that usage of these features in the existing DL implementation can cause unbounded tardiness with AC. For the remaining features, though usage may not cause unbounded tardiness with AC, we demonstrate that usage can lead to other undesirable effects, detailed later.

Second, we show that the existing DL implementation is broken under SP affinities in the sense that unbounded tardiness may result in under-utilized systems. We present a patch that modifies DL’s migration logic and AC such that tardiness is bounded for all such systems. This patch is intended for Linux 5.4.69, as 5.4 was the most recent LTS release at time of writing; we generally refer to this version as the current implementation unless specified otherwise, though the details discussed in this work do not seem to have changed with recent releases. This patch was designed to be as minimally invasive as possible such that the development of existing features would not be hindered by including this patch.

Third, we provide a high-level overview of our proof of soundness for our modified AC under an abstraction of our patched DL (due to space constraints, the full proof is deferred to App. A). Our proof is based on existing proof techniques from [16]. Our proof modifies these techniques because SAPA-EDF satisfies an invariant that our patched DL does not. Our patched scheduler maintains a weaker invariant that we show is sufficient for bounded tardiness under SP scheduling. Our patched DL compromises between schedulers proposed in theory, which migrate tasks aggressively, and DL, which is unable to make guarantees.

**Organization.** The rest of this paper is organized as follows. After covering needed background in Sec. 2, we present and categorize the list of DL features not considered by [6] in Sec. 3. We demonstrate the non-optimality of DL under SP affinities and present our patch in Sec. 4. We provide an overview of the proof of correctness of this patch under an idealized abstraction of our patched DL in Sec. 5 (again, the formal proof is deferred to an appendix). We evaluate the overheads of our patch relative to the original implementation in Sec. 6. We conclude in Sec. 7.

## 2 BACKGROUND

We start by presenting our system model, mostly derived from [15] with some modifications to consider dynamic task systems. We then discuss how DL fits this system model.

### 2.1 Task Model

We consider a system of $N$ implicit-deadline sporadic tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_N\}$ running on $M$ unit-speed CPUs $\pi = \{\pi_1, \pi_2, \ldots, \pi_M\}$. We assume basic familiarity with the sporadic task model. We denote the $j$th job released by task $\tau_i$ as $\tau_{i,j}$, where $j \geq 1$. Job $\tau_{i,j}$ must be completed before job $\tau_{i,j+1}$ is allowed to execute. We let $C_i$ denote the worst-case execution time (WCET) of $\tau_i$ over all its jobs. We let $T_i$ denote the period of task $\tau_i$. The utilization $u_i$ of task $\tau_i$ is given by $C_i/T_i$. We denote the release time, deadline, and completion time of job $\tau_{i,j}$ by $r_{i,j}, d_{i,j}$, and $f_{i,j}$, respectively, where $d_{i,j} = r_{i,j} + T_i$ (implicit deadlines).

At time $t$, a job $\tau_{i,j}$ is either unreleased ($t < r_{i,j}$), pending ($r_{i,j} \leq t < f_{i,j}$), or complete ($t \geq f_{i,j}$). If a task $\tau_i$ has pending jobs at $t$, then its ready job at $t$ is its earliest-released pending job at $t$.

For a job $\tau_{i,j}$, its **response time** is given by $f_{i,j} - r_{i,j}$, and its tardiness by $\max(0, f_{i,j} - d_{i,j})$. The tardiness of task $\tau_i$ is the supremum of the tardiness of its jobs. If the tardiness of all tasks in $\tau$ is bounded under a given scheduler, then $\tau$ is **soft real-time (SRT)**-schedulable under that scheduler. $\tau$ is **SRT-feasible** if it is SRT-schedulable under some scheduler. A scheduler is **SRT-optimal** if any SRT-feasible task system is SRT-schedulable under it. Analogous terms apply to hard real-time (HRT) systems, in which no tasks can be tardy. Feasible without a prefix denotes both SRT- and HRT-feasible.

At any time, a task is either **active** or **inactive**. A task is initially inactive, and must become active in order to release jobs. Each task becomes active at most once, and after becoming inactive, remains so indefinitely (this is for simplicity of notation, as a task becoming active a second time is equivalent to an unrelated task becoming active). At time $t$, the set of active tasks is denoted $\tau(t) \subseteq \tau$, and the set of CPUs they may execute upon is denoted $\pi(t) \subseteq \pi$. The set of tasks partitioned onto CPU $\pi_j$ is denoted $\tau(t, \pi_j) \subseteq \tau(t)$.

These functions are defined such that any changes that occur at time instant $t$ are reflected in $\tau(t), \pi(t)$, and $\tau(t, \pi_j)$.

### 2.2 From Threads to Tasks

Linux threads may not adhere to the sporadic task model. Thus, any Linux thread under DL is encapsulated in a Constant Bandwidth Server (CBS). A CBS is conceptually similar to a sporadic task, with a maximum budget $Q$, period $T$, and bandwidth $Q/T$ that are

1Note that "active" and "inactive" in this work are distinct from their definitions in DL’s documentation and code.
With this assumption, a CBS behaves identically to a sporadic task, was not considered in [6] and [16]. If $t < d$, then the CBS deviates from the sporadic task model such that per-job tardiness is well-defined, the actual implementation does not make jobs explicit.

Fig. 1 describes the state transitions of a CBS. While a CBS encapsulates a DL thread, it is either eligible, throttled, or idle. The replenish($t$) function in Fig. 1 denotes that $q \leftarrow Q$ and $d \leftarrow t + T$. sched_yield denotes that the encapsulated thread called the sched_yield system call, which is used in DL to indicate that the current job has completed without exhausting its budget.

At the time $t$ when a thread first enters DL, its encapsulating CBS begins in the eligible state at time $t$. Its deadline is then set to $d \leftarrow t + T$ and its budget to $q \leftarrow Q$. This corresponds to the release of the first job. A CBS is eligible to be scheduled while and only while in the eligible state. It is scheduled alongside other servers with priority $d$. While the CBS is eligible, its budget $q$ decreases at unit rate when it is scheduled and is unchanged otherwise.

For now, assume that at any time $t$ when a CBS wakes, $t \geq d$. This precludes the dotted transition from the idle to eligible state. With this assumption, a CBS behaves identically to a sporadic task, with transitions into (resp., out of) the eligible state corresponding to job releases (resp., completions). Jobs are separated periodically when the CBS is throttled (the CBS is replenished at the deadline $d$ and we assume implicit deadlines), while sporadic separations are implemented by suspending (the CBS is replenished at the time of waking, $t$, and we assume $t \geq d$).

If at the time $t$ a CBS wakes and $t < d$, then the CBS deviates from the sporadic task model considered in [6] and [16]. If $t < d$ and $q/(d - t) \leq Q/T$, the CBS returns to the eligible state while keeping the prior job’s budget and deadline (note the dotted transition does not replenish). This is analogous to a job self-suspending, which was not considered in [6] and [16]. If $t < d$ and $q/(d - t) > Q/T$, then the CBS is replenished at $t$. Because we have assumed implicit deadlines and $t < d$, this means the separation between consecutive jobs of the CBS was less than $T$, which is also forbidden by the sporadic task model. The impacts of these differences between CBSs and sporadic tasks on tardiness is discussed in more detail in Sec. 3. For reasons discussed in Sec. 3, we assume that $t \geq d$ whenever a CBS wakes, thereby removing the dissimilarities between CBSs and sporadic tasks. Because they have identical semantics under this assumption, we refer to CBSs and their encapsulated threads as tasks unless specified otherwise.

### 2.3 CPUSets and AC

A task’s WCET, period, and relative deadline are provided as arguments when it first enters DL from another scheduling class via the sched_setattr system call.

DL schedules tasks using clustered EDF, wherein each task assigned to a cluster is only permitted to migrate between CPUs within said cluster. Both global and partitioned EDF are special cases of clustered EDF. In Linux, clusters are created and managed via the cpuset subsystem under the cgroup virtual file system (a cgroup is Linux’s notion of a cluster). Tasks and CPUs are added to and removed from a cpuset dynamically with the cpuset subsystem. $\tau(t)$ and $\pi(t)$ in Sec. 2.1 are abstractions for the set of DL tasks and CPUs assigned to a cpuset by the cpuset subsystem.

AC prevents the starvation of non-DL workloads and guarantees bounded tardiness to DL tasks within a cpuset. Generally, AC maintains the invariant that for any cpuset,

$$
\forall t: \sum_{t \leq t(t)} u_i \leq \frac{\text{sched_rt_runtime}_{us}}{\text{sched_rt_period}_{us}} [\pi(t)],
$$

where sched_rt_runtime$_{us}$ and sched_rt_period$_{us}$ are set with the proc virtual file system and have default values of 950000 and 100000, respectively. sched_rt_runtime$_{us}$ should always be set to be less than sched_rt_period$_{us}$. If so, (1) guarantees non-DL workloads in the cpuset are given time to execute because the system is not over-utilized [6], though this guarantee may be invalid for reasons discussed in Sec. 3.

AC must verify that (1) is maintained on changes to $\tau(t)$ (tasks entering DL), changes to $\pi(t)$ (changes to the CPUs in the cpuset), and changes to sched_rt_period$_{us}$ or sched_rt_runtime$_{us}$. Requests for changes fail if doing so would violate (1) for any cpuset. These changes occur immediately otherwise.

The summation in (1) must be decreased when a thread leaves DL. Though the thread may leave immediately upon request, its server must remain until an event called its 0-lag time passes. This implementation detail exists to support HRT guarantees. As we are concerned with SRT, we do not consider this rule in our model.

### 2.4 Fine-Grained Affinities

In Linux, the sched_setaffinity system call is an auxiliary method to the cpuset subsystem of setting affinities. sched_setaffinity refers to the cpuset subsystem in that calls that contain CPUs outside of a task’s cpuset will ignore said CPUs. Additionally, when $\pi(t)$ is modified, all tasks in the cpuset gain affinity for all CPUs in $\pi(t)$. This overwrites any changes made with sched_setaffinity, even if the modified set of CPUs is a superset of the original.

The usage of sched_setaffinity is not permitted for DL tasks unless AC is disabled by writing -1 to sched_rt_runtime$_{us}$. AC must be disabled as bounded tardiness to tasks is not guaranteed under arbitrary affinities.

---

2The behavior of a CBS differs slightly if deadlines are not implicit.

3Though the cpuset subsystem actually organizes cpusets hierarchically, DL requires that cpusets with DL tasks to be exclusive from each other, which we also assume.

4These file names derive from SCHED_RT, whose accounting code DL intertwines with. We assume no RT tasks to focus on DL.
3 PROBLEMATIC FEATURES

Since being merged into the kernel, the list of features supported by DL has far outgrown the simplistic system model in [6]. Unfortunately, many of these features have been implemented in ways that break the bounded tardiness properties AC is supposed to guarantee, necessitating that we omit them from the system model assumed in our SP DL patch. When practical, we present scheduling traces of real DL tasks produced by trace-cmd and visualized with kernel shark to validate claims about the scheduler’s behavior. The scripts that create these traces are available online [14].

3.1 Features that Break Bounded Tardiness

**Dynamic tasks.** Technically, a task’s WCET, period, and relative deadline are dynamic and can be changed with the sched_setattr system call. DL will immediately enact any change in parameters requested with sched_setattr so long as the task’s resulting utilization does not violate (1). While the task’s static parameters and bandwidth are changed immediately, the task’s current utilization remains the same. This can be exploited as follows to result in unbounded tardiness.

> **Ex. 1.** (Corresponds with Fig. 2.) We assume the tasks in this example never suspend. Consider a cpuset containing one CPU \( \pi_1 \) and two CPUs \( \pi_2 \) and \( \pi_3 \) and \( \pi_4 \) with no tasks. At time \( t = 0 \), task \( \tau_1 \) requests to enter DL with \((C_1, T_1) = (94, 100)\). This task is accepted because doing so will not violate (1). \( \tau_1 \) releases its first job at time 0 with \( C_{1,1} = 94 \) and \( d_{1,1} = 100 \), before immediately requesting \( C_1 \) be changed to 1. \( C_{1,1} \) and \( d_{1,1} \) remain as 94 and 100 after this change. At time \( t = 10 \), task \( \tau_2 \) requests to enter with \((C_2, T_2) = (94, 100)\). This request is accepted by AC because \( \tau_1 \) reduced its utilization. \( \tau_2 \) releases its first job with \( C_{2,1} = 94 \) and \( d_{2,1} = 100 \). After this point, both tasks release jobs periodically.

However, prior to when \( \tau_1 \) returns from the throttled state (see Fig. 1), \( \tau_2 \) changes \( C_2 \) to 1 and \( \tau_1 \) changes \( C_1 \) to 94. When the next job of \( \tau_1 \) becomes ready at \( t = 100 \), this job’s budget and deadline are determined entirely by \( \tau_1 \)’s parameters at the instant it becomes ready. Thus, the budget of the next job is set to 94. Likewise, prior to when \( \tau_2 \) is replenished at \( t = 188 \) (recall from Fig. 1 that a tardy task is replenished once its prior job completes), \( \tau_1 \) sets \( C_1 \) to 1 such that \( \tau_2 \) can set \( C_2 \) to 94. As the total utilization of the system technically never exceeds 0.95, all requests are accepted by AC.

If \( \tau_1 \) and \( \tau_2 \) continue taking turns with having \( C = 94 \), then every released job in this system has parameters as if both tasks always had utilizations of 0.94. Because the CPU only has a capacity of 1.0, the system is over-utilized and results in unbounded tardiness even though (1) is never violated.

**Defect 1. Dynamic tasks can break AC.**

**Mitigation 1.** We reject requests to modify DL tasks’ parameters. Any change in parameters after server creation must be implemented by leaving DL and reentering with a fresh server.

**DVFS.** The goal of DVFS is to reduce power consumption by scaling down CPU frequency. DVFS affects DL because DL must ensure that frequencies remain high enough that some level of real-time performance guarantee is maintained for tasks. DL must also account for reduced CPU frequency when deallocating tasks’ budgets. This is managed in DL with the GRUB-PA system.

We refer to [13] for a full description of GRUB-PA, which is out of the scope of this work. At a high level, at all times, GRUB-PA tracks the total bandwidth of DL tasks on each CPU’s runqueue in a variable denoted \( U_{act}(t) \). If \( U_{act}(t) < 1 \), then the frequency of the corresponding CPU is scaled by \( U_{act}(t) \) of its maximum frequency and the budget of any task executing on said CPU is depleted at rate \( U_{act}(t) \) (tasks’ WCETs are assumed to be provisioned based on CPUs running at maximum frequency). Otherwise, the CPU runs with its maximum frequency and the budget of any executing task is depleted at the standard unit rate. This can be exploited as follows to result in unbounded tardiness.

> **Ex. 2.** Consider a cpuset with two CPUs and three tasks with \((C, T) = (63, 100)\). This system is accepted by AC because the total utilization is 1.89 which is less than 95% (recall the default values in (1)) of the capacity of 2 CPUs. However, because any task can be on at most one CPU’s runqueue at any time, there is always a CPU with \( U_{act} \leq 0.63 \). Because CPU frequencies are scaled by \( U_{act} \) and the other CPU is unable to scale frequency beyond its maximum, the actual capacity delivered by the CPUs to DL tasks is at most 1.63 < 1.89, the total utilization. Because the system is over-utilized, at least one task experiences unbounded tardiness.

**Defect 2. DVFS causes unbounded tardiness in DL with AC.**

**Mitigation 2.** We require that DVFS be disabled for AC.

**GRUB-PA** is the latest entry in a long line of GRUB variants implemented in DL. The code for M-GRUB, the GRUB variant used before GRUB-PA, actually still remains in the current DL implementation and can be triggered with the SCHED_FLAG_RECLAIM flag, though we have not considered it in this work because it is incompatible with the more recent GRUB-PA and DVFS. Nevertheless, it has also not been proven that M-GRUB does not violate bounded tardiness under AC, though we conjecture that this is the case.

**Asymmetric capacities.** Support for asymmetric CPU capacities, in which different CPUs complete work at different rates, was added...
to DL in Linux 5.9. The fastest CPU in the system is assigned a capacity\(^5\) of 1.0 and all other CPUs are assigned capacities relative to the fastest CPU. Let \(\text{cap}(\pi_i)\) denote the capacity of CPU \(\pi_i\). AC was also modified in 5.9 such that the r.h.s. of (1) considers \(\sum_{\pi_i \in \pi(t)} \text{cap}(\pi_i)\) instead of \(|\pi(t)|\).

In the real-time literature, a large body of work considers platforms with asymmetric-capacity CPUs. In these works, such platforms are often called uniform multiprocessors. One such work is [18], which presents an SRT-optimal EDF variant for uniform multiprocessors. DL’s capacity-aware version of (1) admits a superset of the systems accepted by the feasibility condition in [18], proving that AC permits systems with unbounded tardiness.

**Defect 3.** Asymmetric capacities can cause AC to admit task systems with unbounded tardiness.

**Mitigation 3.** We assume CPUs are identical.

### 3.2 Features that Negatively Affect Tardiness

For the following features, we conjecture that their usage does not break bounded tardiness under AC. Nevertheless, though these features may not technically break AC, they can have undesirable effects on tardiness. We chose to omit these features from our system model due to such negative effects and difficulties with the complexities caused by considering them in proofs.

**Inter-cpuset migration.** A dynamic behavior of DL not considered in [6] is the migration of a task from one cpuset to another. Such a migration will be accepted by AC so long as doing so will not violate (1) for the target cpuset. When the migration occurs, the task’s utilization is immediately subtracted from its original cpuset and added to that of its target. The server may keep its prior budget and period. According to the documentation [2], the server and task should have the same deadline in this case.

\*\*Ex. 3.\* (Corresponds with Fig. 3.) Consider two cpusets, one with a single CPU \(\pi_1\) and a task \(\tau_5\) with \((C_5, T_5) = (2, 10)\) and another with three CPUs \(\pi_2, \pi_3,\) and \(\pi_4\) and four tasks \(\tau_1, \tau_2, \tau_3,\) and \(\tau_4\) all with \((C, T) = (70, 100)\).

At time \(t = 92\), \(\tau_4\) requests to migrate to the cpuset containing \(\pi_1\) and \(\tau_5\). This is accepted by AC as \(\tau_4\) and \(\tau_5\)’s combined utilization is less than 0.95. However, \(\tau_4\) keeps its current deadline and budget when it migrates, thereby causing \(\tau_5\) to become tardy when it otherwise would not have been.

**Obs. 1.** Inter-cpuset migration can cause higher tardiness than expected under DL.

**CBS suspensions.** As discussed in Sec. 2.2, a CBS may not adhere to the sporadic task model considered in [6] by suspending and resuming before a job’s deadline, resulting in either a self-suspending job or a separation between CBS job releases being less than a period. It is not obvious that this does not break bounded tardiness under AC, as arbitrary self-suspendations are known to cause capacity loss [12].

\*\*Ex. 4.\* (Corresponds with Fig. 4.) Consider a cpuset with two CPUs \(\pi_1\) and \(\pi_2\) and three servers \(\tau_1, \tau_2,\) and \(\tau_3\) all with \((C, T) = (63, 100)\).

This is not the only nuance caused by CBS and suspensions. Suspending, even for infinitesimally short durations, can cause a server to execute at a rate less than its bandwidth, as shown in the following example. This kind of behavior can be especially problematic when the encapsulated workload of the server is a real-time task whose WCET and period are the same as the server’s maximum budget and period. According to the DL documentation [2], the server and task should have the same deadline in this case.

\*\*Ex. 4.\* (Corresponds with Fig. 4.) Consider a cpuset with two CPUs \(\pi_1\) and \(\pi_2\) and three servers \(\tau_1, \tau_2,\) and \(\tau_3\) all with \((C, T) = (63, 100).\) However, the encapsulated real-time task of \(\tau_3\) must suspend briefly...
at the end of every job, perhaps to do some I/O. Meanwhile, $r_1$ and $r_2$ only suspend between jobs when none are available to execute.

All servers release jobs at $t = 0$, with $r_1$ and $r_2$ winning the deadline tie. This causes $r_3$ to complete its job after its deadline, after which it suspends for a small time interval. However, when it returns from the idle state, its deadline is updated based on the time of waking instead of its previous deadline (recall Fig. 1). This causes the release pattern of $r_3$ to become non-periodic, even if the encapsulated workload is periodic. For example, repeating the release pattern in Fig. 4a causes $r_3$ to execute for 63 time units around every 120 time units, even though its underlying workload requires its execution every 100 time units.

Obs. 2. An encapsulated real-time task whose jobs’ may self-suspend can be starved under DL, even in a CBS with bandwidth equal to the task’s utilization.

The task’s deadline falls behind the server’s deadline because the server “cheats” by interpreting any suspension after its own deadline to be a suspension between jobs (meaning that the encapsulated task has no jobs available), when in actuality the task is self-suspending within a job. At a high level, this is why self-suspending tasks on a CPU remain tardy even when they do not under CBS.

As far as we are aware, how to determine DL server parameters for encapsulating self-suspending real-time tasks such that server and task deadlines will stay synchronized has not been addressed.

4 ADDING SP SCHEDULING TO DL

We describe our proposed patch in four subsections, each addressing a distinct aspect of the implementation: bypassing the throttled state, pushing to the latest CPU, AC, and dynamic affinities. We begin each subsection by demonstrating how each aspect causes problems in SP systems. Next, we summarize how this behavior arises in the current implementation. Last, we explain how our patch modifies the implementation. Our patch is available online [14].

4.1 Bypassing the Throttled State

Problem. In DL, a task that completes a job by exhausting its budget or calling sched_yield after its deadline will remain eligible, bypassing the throttled state (as in Fig. 1). If so, this task continues executing its next job on the same CPU if it is not preempted by a different task with an earlier deadline. This is desirable under clustered scheduling because a task that continues to execute on the same CPU retains its cache hotness. This behavior can cause unbounded tardiness when partitioned tasks exist.

* Ex. 5. (Ex. 4 of [15] corresponds with Fig. 5.) Consider a cpuset with CPUs $\pi_1$, $\pi_2$ and $\pi_3$ with tasks $\pi_1$, $\pi_2$, $\pi_3$, $\tau_1$, $\tau_2$, and $\tau_3$. Let ($C_1$, $T_1$) = ($C_3$, $T_3$) = (2, 6), ($C_2$, $T_2$) = ($C_4$, $T_4$) = (2, 2), and ($C_5$, $T_5$) = (1, 6), $\tau_1$, $\tau_3$, and $\tau_5$ are partitioned on $\pi_1$, $\pi_2$, and $\pi_3$, respectively, while $\tau_2$ and $\tau_4$ are migrating.

$\tau_2$ and $\tau_4$ release jobs periodically. Initially, $\tau_2$ and $\tau_4$ execute on $\pi_1$ and $\pi_3$, respectively. At $t = 6$, fixed tasks $\tau_1$ and $\tau_5$ preempt $\tau_2$ and $\tau_3$, respectively. The only other processor available to both $\tau_2$ and $\tau_4$ is $\pi_2$ ($\tau_2$ cannot preempt $\tau_5$ on $\pi_3$ and $\tau_4$ cannot preempt $\tau_1$ on $\pi_1$), which they cannot both use. We assume the tiebreak here favors $\tau_2$ and it is scheduled, while $\tau_4$ does not execute until $t = 8$ when it resumes execution on $\pi_1$. $\tau_2$ is also forced to migrate off $\pi_2$ by fixed task $\tau_5$ at $t = 12$. This repeats at $t = 18$, except here $\tau_4$ is scheduled over $\tau_2$ because it is tardy by 2.0 time units due to not being scheduled over [6, 8]. As a result, $\tau_2$ also becomes tardy by 2.0 time units by $t = 20$. This pattern can repeat indefinitely, and with each occurrence, the maximum tardiness experienced by either $\tau_2$ or $\tau_4$ increases by 2.0.

Implementation. Explaining why tasks migrate as in Ex. 5 and giving context to this and later subsections requires an explanation of DL’s migration code. Because a task can only run on a given CPU while on said CPU’s dl_rq (the DL-specific runqueue data structure), tasks must be exchanged between dl_rq’s to migrate.

The operations for exchanging tasks between dl_rq is called pull (from all other CPUs to the pulling CPU) and push (from the pushing CPU to a target CPU). Pulls are used to emulate a per-cluster runqueue during scheduling decisions. Prior to any scheduling decision, a CPU pulls the unscheduled task with earliest deadline and affinity for said CPU from each other CPU’s dl_rq. This allows the scheduling CPU to select the task with earliest deadline from any dl_rq in the cpuset as if they were stored in a single runqueue.

Pushes occur when a task is preempted or becomes eligible. In a push, the pushing CPU sends the unscheduled migrating task with earliest deadline on its dl_rq to the dl_rq of a target CPU.

Let the deadline of a CPU be the deadline of the eligible task with earliest deadline on its dl_rq. The target CPU of a push is either a CPU with no DL tasks on its dl_rq (called a free CPU) or, if the cpuset has no free CPUs, the CPU with the latest deadline among the other CPUs in the cpuset. Note that the latest CPU may be the pushing CPU, in which case the task stays on the same dl_rq.

A task that exhausts its budget or calls sched_yield is dequeued from its CPU’s dl_rq. What occurs next depends on whether this task bypasses the throttled state (its deadline has passed) or enters it (its deadline is in the future).

A task that bypasses the throttled state is immediately enqueued back onto its CPU’s dl_rq with replenished budget and updated deadline. If another task on this CPU’s dl_rq now has an early enough deadline to preempt the replenished task, it does so. Otherwise, the replenished task continues executing on the same CPU.
Recall that tasks bypassing the throttled state and continuing to execute on the same CPUs resulted in unbounded tardiness in Ex. 5. For example, at its job boundaries at times 2 and 4 in Fig. 5, \( \tau_2 \) does not migrate to free CPU \( \pi_2 \) even though its current CPU \( \pi_1 \) has a partitioned task \( \tau_1 \) with a ready job. \( \tau_2 \) does not migrate at these job boundaries because it bypasses the throttled state.

On the other hand, a task that enters the throttled state is unscheduled from the CPU that runs it. When this task returns from the throttled state to the eligible state (in function \( \text{dl_task_timer} \), an \( \text{hr_timer} \) callback), it is enqueued back onto this CPU’s \( \text{dl rq} \) with a replenished budget and updated deadline. Because the replenished task became eligible, this CPU attempts to push a thread.

**Patch.** Ex. 5 would not have unbounded tardiness if successive jobs of tasks \( \tau_2 \) and \( \tau_1 \) would migrate when a free CPU or CPU with later deadline is available, thereby allowing partitioned tasks to execute. For example, if \( \tau_2 \) had migrated to free CPU \( \pi_2 \) at time 2 instead of 6 in Fig. 5, then the first job of \( \tau_1 \) would have been able to execute on \( \pi_1 \) over the interval \([2, 4]\). This would have freed \( \pi_1 \) over \([6, 8]\), allowing migrating task \( \tau_2 \) to execute and preventing its tardiness. Recall that successive jobs of tardy tasks do not migrate because they bypass the throttled state, skipping the push that occurs when a task returns from the throttled state.

Our patch addresses this by removing the branch in which a task bypasses the throttled state. This causes successive jobs of tardy tasks that would have otherwise continued to execute on the same CPU to be pushed from that CPU due to the tardy task becoming eligible from the throttled state. For example, at time 2 in Fig. 5, \( \tau_2 \) completes its job. Under our patch, \( \tau_2 \) would be throttled and immediately unthrottled because its next job already ready at time 2 (instead of not being throttled at all, as in the original implementation). Due to being unthrottled, \( \tau_2 \) would be pushed from \( \pi_1 \) to free CPU \( \pi_2 \). This is the schedule described in the previous paragraph that reduces tardiness.

Note that callback function \( \text{dl_task_timer} \) will not push a scheduled task, as it is not safe to migrate an executing task. This is problematic because throttled tasks may still be scheduled because rescheduling is not instantaneous. To guarantee that a tardy task is pushed, \( \text{dl_task_timer} \) must wait for the tardy task to be unscheduled. Unfortunately, it does not help to wait within \( \text{dl_task_timer} \) for the relevant CPU to reschedule, as both \( \text{dl_task_timer} \) and the rescheduling code both must acquire this CPU’s \( \text{rq} \) lock (the \( \text{rq} \) struct contains generic runtime fields as well as scheduler-specific data structures such as the \( \text{dl rq} \)). In our patch, when \( \text{dl_task_timer} \) executes and observes that the task to be pushed is still scheduled, the callback releases the \( \text{rq} \) lock and retries in the future, giving the relevant CPU the chance to reschedule. This is done with \( \text{hr_timer_forward_now} \).

To validate that our patch forces tasks to be pushed, consider the traces in Fig. 6. In Fig. 6a, the topmost task attempts to migrate whenever it completes a job (the vertical lines in the trace). Meanwhile, in Fig. 6b, this same task does not migrate unless preempted.

### 4.2 Pushing to the Latest CPU

**Problem.** The target CPU of a push is the CPU with latest deadline. In DL, the deadline of the pushing CPU may be from the task being pushed. This can result in priority inversions under SP scheduling.

**Ex. 6.** (Corresponds with Fig. 7) Consider a cpuset with two CPUs \( \pi_1 \) and \( \pi_2 \) and three tasks \( \tau_1, \tau_2, \) and \( \tau_3 \) with \((C_1, T_1) = (10, 70), (C_2, T_2) = (10, 50), \) and \((C_3, T_3) = (5, 10)\). \( \tau_1 \) and \( \tau_2 \) are partitioned on \( \pi_1 \) and \( \pi_2 \), respectively, while \( \tau_3 \) is migrating. \( \tau_3 \) releases its first job at \( t = 0 \) and executes on CPU \( \pi_3 \) until \( t = 5 \), at which point \( \tau_3 \) is throttled. At time \( t = 7 \), both \( \tau_1 \) and \( \tau_2 \) release their first jobs and begin executing. At time \( t = 10 \), \( \tau_1 \) returns from the throttled state (recall Fig. 1). This results in \( \tau_1 \) being placed back onto \( \pi_1 \)’s \( \text{dl rq} \) (as \( \pi_1 \) was \( \tau_1 \)’s last CPU) and \( \tau_1 \) attempting to push \( \tau_3 \).

Because, at the instant \( \tau_3 \) is pushed, \( \tau_1 \) executes \( \tau_1 \) with deadline 77 and \( \tau_2 \) executes \( \tau_2 \) with deadline 57, \( \tau_3 \) should remain on \( \pi_1 \) and preempt \( \tau_1 \), whose deadline is later than that of \( \tau_2 \). The actual behavior exhibited by DL is that \( \tau_1 \) will push \( \tau_3 \) to \( \pi_2 \), preempting \( \tau_2 \). This is confirmed in Fig. 7b.

**Implementation.** The search for the target CPU is facilitated by a per-cpuset cpu dl struct. A cpu dl struct keeps a bitmask of the free CPUs in the cpuset and organizes the other CPUs into a max heap by deadline. This bitmask and heap are updated with functions inc dl deadline and dec dl deadline, which are called whenever a task is enqueued or dequeued off a CPU’s \( \text{dl rq} \), respectively.
In order to be pushed to a target CPU, a task must be on the pushing CPU’s d1_rq. Because this task was enqueued onto this d1_rq, the cpudl heap considers the pushed task’s deadline when evaluating the pushing CPU’s deadline. This may cause the cpudl to identify another CPU as a push target, even if the other tasks on the pushing CPU’s d1_rq all have later deadlines than this target CPU. This is not a problem under clustered scheduling because a task that is wrongfully preempted on the target CPU can compensate by migrating to the pushing CPU, but this is not an option for partitioned tasks such as in Ex. 6.

After a target CPU is identified with cpudl, the rq lock of the target CPU is acquired with function double_lock_balance (push_d1_task, the function that implements pushes in DL, is already holding the rq lock of the pushing CPU). The pushed task is then dequeued from the pushing CPU’s d1_rq and enqueued onto that of the target CPU. Finally, both CPU’s rq locks are released.

**Patch.** The cpudl would not mistakenly target the wrong CPU in a push if it did not consider the deadline of the task being pushed. Our patch accomplishes this by dequeuing any task in the process of being pushed before the cpudl is accessed to determine the target CPU (in function find_later_rq). Dequeuing ahead of accessing the cpudl ensures that dec_dl_deadline is called prior to identifying a target CPU, preventing the pushing CPU from being represented by the pushed task’s deadline in the cpudl.

Pushes may fail due to race conditions that seem fundamental in DL, as rq locks may be released in double_lock_balance in order to acquire them in order (to prevent deadlock). Care must be taken so that an inconsistent state is not exposed to concurrently executing kernel code during a push while no rq lock is held. For this reason, we immediately enqueue a pushed task back onto the pushing CPU’s d1_rq once find_later_rq (the function that checks the cpudl) returns. This enqueue is redundant if a target CPU is successfully identified and the push does not fail due to race conditions, as the task must then be dequeued once again to be enqueued onto the target CPU’s d1_rq.

### 4.3 AC

**Problem.** The existing AC does not prevent a user from overloading a given CPU in a cpuset by partitioning tasks with combined utilization exceeding 1.0 onto that CPU.

**Implementation.** As discussed in Sec. 2.3, the existing AC maintains (1) for all cpusets. Each cpuset in DL is represented by a root_domain struct (e.g., the cpudl is a field of the root_domain). Each root_domain contains a pointer to a d1_bw struct that stores the total utilization of all DL tasks assigned to the corresponding cpuset (i.e., the summation in (1)) in its total_bw field. AC rejects any requests that falsify (1). Such requests may include adding tasks to τ(t), changing the CPUS in π(t), or modifying either of sched_rt_runtime_us or sched_rt_period_us. (1) is checked in function sched_dl_global_validate.

Adding a task to τ(t) using sched_setattr will fail if the server to add does not have affinity for every CPU in π(t). This is checked by comparing field span in the root_domain (bitmask of CPUS in the cpuset) and field cpus_ptr in the task’s task_struct (bitmask of CPUs the task has affinity for) in helper function __sched_setscheduler. This check is skipped when AC is disabled.

**Patch.** Partitioned tasks are created by giving a task affinity for a single CPU in its cpuset prior to entering DL (as discussed in the next subsection, DL tasks are not allowed to change their affinities). We modify __sched_setscheduler so that sched_setattr will not fail if a task has affinity for a single CPU.

Besides the condition in (1), we modify AC to also maintain,

\[ \forall t: \forall \pi_j \in \pi(t): \sum_{\tau_i \in \tau(t, \pi_j)} u_i \leq \frac{\text{sched_rt_runtime_us}}{\text{sched_rt_period_us}} \]

This is done by adding checks for (2) where __dl_overflow is called and in sched_dl_global_validate, which can cause (2) to be violated by adding a new task to τ(t, π_j) or modifying its r.h.s. Similar to how the l.h.s. of (1) is tracked in total_bw stored in the root_domain, the l.h.s. of (2) is tracked in a new variable partitioned_bw stored in each π_j’s d1_rq.

(2) is not checked in dl_cpuset_cpumask_can_shrink, which is called when π(t) changes. This is because the expected behavior under Linux when the CPUS in a cpuset are changed is that any affinity changes made with sched_setaffinity are lost. Thus, all partitioned DL tasks in the cpuset will become migrating, making (2) irrelevant. Because all partitioned tasks become migrating, partitioned_bw must be set to 0 for any CPUS in the prior π(t).

### 4.4 Dynamic Fine-Grained Affinities

**Problem.** We neglected to mention any checks made for (2) when sched_setaffinity is used to change a partitioned task’s CPU, nor did we mention how we modified sched_setaffinity to accept such requests. As it turns out, allowing DL tasks to dynamically change their affinity can actually starve such tasks.
Ex. 7. This example corresponds with Fig. 8. Consider a SP system on a cpuset with two CPUs \( \pi_1 \) and \( \pi_2 \) and two tasks \( \tau_1 \) (initially partitioned on \( \pi_1 \)) and \( \tau_2 \) migrating with \( (C_1, T_1) = (1, 2) \) and \( (C_2, T_2) = (1, 1) \). Both tasks release jobs at time \( t = 0 \). \( \tau_2 \), whose deadline is earlier than \( \tau_1 \)'s deadline, executes on \( \pi_1 \) until preempted by \( \tau_1 \) at \( t = 2 \). \( \tau_2 \) migrates to \( \pi_2 \) once preempted. However, \( \tau_1 \) requests to change its affinity from being partitioned on \( \pi_1 \) to \( \pi_2 \). Similar to how a task's deadline is updated based on the current time when returning from the idle state after said task's original deadline (recall Fig. 1), \( \tau_1 \)'s deadline is updated from 2 to 4 when it migrates at \( t = 2 \). Thus, \( \tau_1 \) does not have an early enough deadline to preempt \( \tau_2 \), and so it continues to be unscheduled.

Repeating this pattern of dynamic affinity requests can prevent \( \tau_1 \) from executing indefinitely, as in Fig. 8a (the reason that the workload corresponding to \( \tau_1 \) in Fig. 8b executes briefly every two time units is because the request to change affinity does not occur instantaneously at the moment of preemption).

Implementation. When AC is enabled, \texttt{sched_setaffinity} performs the same check between the cpuset’s CPUs and the requested affinity as \texttt{__sched_setscheduler}. Thus, effectively \texttt{sched_setaffinity} did nothing for DL tasks, as such tasks needed to already have affinity for all the cpuset’s CPUs to enter DL. When AC is disabled, this check is bypassed and the task is given the requested affinity. If the task no longer has affinity for the CPU whose rq the task is currently on, it is immediately moved to the rq of a CPU in it has affinity for. If the task is tardy during this migration, it is immediately replenished based on the current time \( t \). This sets the task's deadline \( d \leftarrow t + T \), lowering the task’s priority.

Patch. We forbid DL tasks from changing their affinities. We do this by modifying \texttt{sched_setaffinity} to automatically reject any requests for DL tasks. If a user desires to change the affinity of a DL task, the task must first leave DL, change its affinity as a non-DL workload, and reenter DL with a new server. This new server’s deadline will be set based on its time of creation. This ultimately has the same effect as the original implementation, but forcing tasks to leave DL emphasizes to the user that real-time performance guarantees may be invalid under changes to affinity.

Rejecting all requests to \texttt{sched_setaffinity} may be heavy handed, but it is non-trivial to determine what restrictions are necessary to both prevent race conditions and account for such requests in proofs of bounded tardiness.

Altogether, our patch is fairly minor, modifying roughly 200 LOC (for context, the main DL file is roughly 3000 LOC).

5 SOUNDNESS ARGUMENT FOR AC

Recall from Sec. 1 that EDF variant SAPA-EDF was proven to be SRT-optimal in [16], but was not implemented in DL due to its increased migrations and scheduler complexities. Our DL variant is a compromise between SAPA-EDF and the existing DL implementation for SP systems. The soundness of our modified AC derives from the behaviors of SAPA-EDF that our patched DL emulates. We begin with an overview of these behaviors.

SAPA-EDF is SRT-optimal because it prevents \textit{affinity-related priority inversions (ARPI)}. Under SP systems, an ARPI occurs when an unscheduled partitioned task is unable to preempt a migrating task on the partitioned task’s CPU; meanwhile, some other CPU in the same cpuset is free or executes a task with later deadline than the partitioned task. This is a priority inversion because a higher-priority task is left unscheduled in favor of a lower-priority task or free CPU. SAPA-EDF prevents such situations by forcing migrating tasks to migrate to later CPUs whenever ARPIs would have otherwise occurred. While SAPA-EDF prevents ARPIs, our patched DL guarantees that they are \textit{transient}.

Ex. 8. Consider a time instant such that an ARPI exists between partitioned task \( \tau_1 \), migrating task \( \tau_2 \) executing on \( \tau_1 \)'s CPU, and task \( \tau_3 \) with later deadline than \( \tau_1 \).

Under our patched DL, this priority inversion lasts for at most the execution time of \( \tau_2 \)'s job. When \( \tau_2 \)'s job completes, \( \tau_2 \) will always be throttled (recall from Sec. 4 that all tasks that complete jobs are throttled in our patch), and will migrate to \( \tau_3 \)'s CPU when it becomes eligible and is pushed (recall that we’ve modified pushes to always target the CPU that executes the task with latest deadline). Note that it is impossible for \( \tau_2 \) to be preempted before its job completes for the duration of the ARPI, as any migrating task will preempt \( \tau_3 \) before \( \tau_2 \) under our patched DL because \( \tau_3 \) has a later deadline. If a partitioned task preempts \( \tau_2 \), then the ARPI ends by definition (recall from the above paragraphs that for an ARPI, it is required that an executing task be scheduled on \( \tau_1 \)’s CPU).

Because these ARPIs are transient (i.e., bounded), any resulting increase in tardiness relative to SAPA-EDF is also bounded. As tardiness is bounded under SAPA-EDF for any feasible system (because SAPA-EDF is SRT-optimal [16]) and because our AC conditions ((1) and (2)) guarantee that the system is feasible [17], it follows that tardiness is bounded under our patched DL and AC. As a reminder, the formal version of this proof is available in App. A.

6 EVALUATION

In this section, we evaluate the performance of our patched DL variant against the original implementation.

6.1 Experimental Setup

Our experiments and the execution traces presented prior to this section were conducted on a 16 CPU Intel Xeon Silver 4110 multiprocessor. Measured workloads were restricted to a cluster composed of eight CPUs, as these compose a single socket and NUMA node, and clusters should reflect the hardware topology in practice. Periodic workloads were generated for these experiments using taskgen [7, 10] and rt-app [1]. SP task systems were created by applying worst-fit packing to the task sets generated by taskgen and determining any unpacked tasks to be migrating.

We are interested in how our modifications to DL’s migration code described in Sec. 4 affect overheads. Changes to overheads are due to forcing tardy tasks to enter the throttled state, thereby requiring that such tasks wait for an \texttt{hrtimer} callback (dl_task_timer) to complete before becoming eligible again, and due to the added dequeue and enqueue operations required for every push. For measuring the additional latency caused by this \texttt{hrtimer} callback, we inserted ftrace event tracepoints into our patched kernel that are triggered whenever a tardy task is forced into the throttled state and when \texttt{dl_task_timer} returns said task to a runqueue. To measure the duration of pushes, we also inserted tracepoints around \texttt{push_dl_task}. To get a more holistic view of how these changes to
migration code affect performance, we also measured the tardiness tasks experience scaled by their periods.

Taskgen is configured such that each generated task set has a total utilization of 7.52. This is slightly below 95% of eight CPUs’ worth of capacity to guarantee that AC will not reject tasks due to potential rounding in taskgen in our patched kernel (AC must be disabled for SP scheduling in the original). We focus on heavily utilized systems as we are primarily interested in SRT and lightly utilized systems produce less, if any, tardiness. We considered task systems composed of 16 and 40 tasks to consider systems with both heavy and light per-task utilizations. Ten different task systems were measured for each number of tasks. Timestamps for each task system were collected over an interval of ten minutes on both the original and our patched kernel.

Latency of forced throttles. Note that we do not compare against the original DL implementation when considering forced task throttling because this overhead is unique to our variant. The distribution of sampled durations during which our patched DL forced a task to be throttled when it would not have in the original implementation is presented in Fig. 9. The average latency caused by a forced throttle was 44 us for systems with 16 tasks and 34 us for systems with 40 tasks. The multimodal distribution of throttle times in Fig. 9a is likely due to our usage of \( \text{hr_timer}_\text{forward} \) \( \text{now} \) with the \( \text{dl_task_timer} \) callback to ensure that tasks forced into the throttle state are unscheduled before \( \text{dl_task_timer} \) attempts to push a task. The time difference between peaks roughly correlates with the time interval the \( \text{hrtimer} \) is forwarded by in our patched DL. As each usage of \( \text{hr_timer}_\text{forward} \) \( \text{now} \) requires the task to wait an additional timer interval, each peak in this histogram likely corresponds with a different number of calls to this function.

These latencies may not be acceptable for servers whose workloads require sub ms response times. However, if the size of these latencies is being caused by waiting for the \( \text{hr_timer} \) as we suspect, an alternative method for forcing tasks that complete jobs to migrate that avoids using the \( \text{hr_timer} \) may be more practical. As stated earlier in Sec. 1, one of the goals of this work was to make as few adjustments to the existing implementation as possible. In future work, we will loosen this restriction.

Duration of pushes. The distribution of sampled push durations for both DL and our patched variant is presented in Fig. 10. These distributions are multimodal because \( \text{push}_{\text{dl_task}} \) contains a retry loop in case the state of the system changes while \( \text{rq} \) locks are dropped in \( \text{double\_lock\_balance} \). The effect of the added enqueue and dequeue operations on push overheads is minor, entailing a change of about 1 us to the average duration of a push.

Tardiness. The distribution of samples of tasks’ tardiness levels is presented in Fig. 11. Average tardiness is negligibly lower under our patched variant compared to the original implementation.

Altogether, the changes made by our patch do not seem to increase overheads by a substantial amount. The most concerning overhead is the latency caused by forwarding the \( \text{dl_task_timer} \) callback function, and even this overhead occurs relatively infrequently. This can be observed by comparing the y-axes of Fig. 9 and 10. The number of pushes performed by the system vastly outnumbers the count of forced throttles.

7 CONCLUSION

We have presented a patch to DL to support semi-partitioned affinities with theoretically sound AC. In looking to prove the soundness of our modified AC, we have highlighted several instances of AC being broken in the existing DL implementation. We believe this has been caused by implementation decisions being driven by heuristics and empirical validation over theoretical analysis. While heuristic-driven development certainly outperforms being oblivious to features such as DVFS or asymmetric capacities, we have demonstrated that intuition is unreliable when it comes to guaranteeing bounded tardiness under AC.

In future work, we would like to consider supporting AC for some of the features we omitted from our model in Sec. 3. We believe that similarly to how our patch compromises between the current DL implementation and SAPA-EDF, a similar compromise can be made between DL and the EDF variants proposed in the theory for asymmetric CPUs.
A FORMAL PROOF OF BOUNDED TARDINESS

We begin by covering some additional notation. We let \( T_{\text{max}} \) (resp., \( C_{\text{max}} \)) denote the maximum period (resp. WCET) over all tasks. We let \( C_i \) denote the required execution time of \( \tau_i \), and denote the spacing between the releases of successive jobs \( \tau_{i,j} \) and \( \tau_{i,j+1} \) of \( \tau_i \) by \( \delta_{i,j} = \tau_{i,j+1} - \tau_{i,j} \), where \( \delta_{i,j} \geq T_i \). We denote the smallest (resp., largest) utilization over all tasks as \( \eta_{\text{min}} \) (resp., \( \eta_{\text{max}} \)). For a set of tasks \( \tau' \subseteq \tau \), let \( \mathbb{U}(\tau') \) denote the sum of the utilizations of tasks in \( \tau' \). Let \( m = \max_{\tau}(\mathbb{U}(\tau)) \).

If task \( \tau_i \) has a ready job \( \tau_{i,j} \) at time \( t \), then we define the functions \( C_i(t) \), \( r_i(t) \), and \( d_i(t) \) to denote to \( C_{i,j}, r_{i,j}, \) and \( d_{i,j} \), respectively. We define the function \( C_i(t) \) to equate to the deadline of the latest completed job of \( \tau_i \) by time \( t \), or 0 if no jobs of \( \tau_i \) have completed. If task \( \tau_i \) has a ready job at time \( t \), then its priority is defined by the priority of its ready job.

### A.1 Deviation Properties

Our proof is a modification of the strategy used in [15] and [16]. We make use of several definitions and lemmas proven in [15] and other works.\(^7\)

**Def. 1.** (Def. 4 of [15]) The virtual time of task \( \tau_i \) at time \( t \geq 0 \) is

\[
\nu(t) = \begin{cases} r_i(t) + T_i \frac{C_i(t) - c_i(t)}{C_i(t)} & \text{\( \tau_i \) has ready job} \\ \max(t, D_i(t)) & \text{else} \end{cases}
\]

**Def. 2.** (Def. 5 of [15]) For task \( \tau_i \) at time \( t \), \( \text{Dev}(\tau_i, t) = u_i(t - \nu(t)) \). For the subset of tasks \( \tau' \), \( \text{Dev}(\tau', t) = \sum_{\tau_i \in \tau'} \text{Dev}(\tau_i, t) \).

**Lemma 1.** (Lemma 1 of [15]) If \( \text{Dev}(\tau_i, t) > 0 \), then \( \tau_i \) has a ready job at \( t \).

**Lemma 2.** (Lemma 2 of [15]) If task \( \tau_i \) has a ready job at time \( t \), then

\[
t - \text{Dev}(\tau_i, t) < d_i(t) \leq t - \frac{\text{Dev}(\tau_i, t)}{u_i} + T_i.
\]

**Corollary 1.** (Corollary 1 of [15]) If for some \( L > 0 \) we have \( \forall t \geq 0 : \text{Dev}(\tau_i, t) \leq L \), then the tardiness of any job of \( \tau_i \) is most \( L/u_i \).

**Lemma 3.** (Lemma 3 of [15]) If at time \( t \), tasks \( \tau_e \) and \( \tau_f \) have ready jobs and we have

\[
\frac{\text{Dev}(\tau_e, t)}{u_e} \geq \frac{\text{Dev}(\tau_f, t)}{u_f} + T_{\text{max}}
\]

then \( d_e(t) < d_f(t) \). (\( e \) signifies “earlier” and \( f \) “later”.)

**Lemma 4.** (Lemma 4 of [15]) For \( \tau_e, e > 0 \), and \( t \geq 0 \), \( \nu(t + e) \geq \nu(t) \).

**Lemma 5.** (Lemma 5 of [15]) For any task set \( \tau' \subseteq \tau \) and time \( t \), if \( \forall \delta > 0 : 3\delta \in (0, \delta] : \text{Dev}(\tau', t + e) > L \) holds, then \( \text{Dev}(\tau', t) \leq L \).

**Lemma 6.** (Lemma 6 of [15]) For arbitrarily small \( e > 0 \), if task \( \tau_i \) is scheduled over \( t, t + e \), then \( \text{Dev}(\tau_i, t + e) \leq \text{Dev}(\tau_i, t) + e(u_i - 1) \).

**Lemma 7.** (Lemma 7 of [15]) For arbitrarily small \( e > 0 \), if task \( \tau_i \) is not scheduled over \( t, t + e \), then \( \text{Dev}(\tau_i, t + e) \leq \text{Dev}(\tau_i, t) + e(u_i) \).

Note that feasibility in the following two lemmas refer to static task systems.

**Lemma 8.** (Lemma 35 of [16]) If a task system \( \tau \) is feasible, then for any set of tasks \( \tau' \subseteq \tau \), the maximum number of CPUs \( M \) that tasks of \( \tau' \) may be simultaneously scheduled upon is such that \( M \geq \mathbb{U}(\tau') \).

For a hypothetical task system \( \tau \) with arbitrary affinities, let \( \alpha(\tau_i) \) denote the set of CPUs task \( \tau_i \) has affinity for. Let \( \alpha(\tau' \tau) \) denote the union of \( \alpha(\tau_i) \) for \( \tau_i \in \tau' \).

**Lemma 9.** (Theorem 1 of [17]) Task system \( \tau \) with arbitrary affinities is feasible if and only if \( \forall \tau' \subseteq \tau : \mathbb{U}(\tau') \leq \alpha(\tau') \).

\(^7\) [15] considered a broader class of schedulers called window-constrained schedulers of which EDF is a special case. Because only EDF is relevant to 0L, the lemmas listed here are special cases of those in [15]. This has the effect of substituting \( \phi \) and \( \chi_{\tau_i}(t) \) in (15) with 0 and \( d_i(t) \).

This is only an excerpt from this theorem.
A.2 Bounded Tardiness under AC

The remainder of this proof establishes that transient affinity-related priority inversions do not cause unbounded tardiness, as discussed in Sec. 5.

**Lemma 10.** Suppose task set \( t' \) is such that for some time \( t' \geq C_{\text{max}} \), for any \( t \in [t' - C_{\text{max}}(t')] \), we have \( t' \subseteq t' \cap t \), tasks of \( t' \) are pending, and tasks of \( t' \) have earlier deadlines than tasks in \( t \). By time \( t' \), a maximal number of CPUs \( M \) execute jobs of \( t' \).

**Proof.** A maximal number of CPUs are executing jobs of \( t' \) when the system contains no affinity-related priority inversions. Recall from the discussion in Ex. 8 that any such affinity-related priority inversions last at most the remaining execution time of the migrating task that executes on the partitioned task’s CPU. This remaining execution time is bounded by \( C_{\text{max}} \). \( \Box \)

**Corollary 2.** Under AC, at time \( t \), the maximum number of CPUs \( M \) that tasks of \( t' \subseteq t \) may be simultaneously scheduled upon is such that \( M \geq U(t') \).

**Proof.** By Lemma 9, and conditions (1) and (2) of AC, \( t' \) would be feasible if it were a static system. By Lemma 8, \( M \geq U(t') \). \( \Box \)

**Corollary 3.** Suppose \( t' \) and \( t' \) are as in Lemma 10 and the system is under AC. Then \( \exists \delta > 0 : \forall \epsilon \in (0, \delta] : \text{Dev}(\tau', t') \geq \text{Dev}(\tau, t' + \epsilon) \) holds.

**Proof.** Let \( \delta \) be arbitrarily small such that any task scheduled (resp., unscheduled) at \( t' \) is scheduled (resp., unscheduled) over \([t', t' + \delta] \). Let \( \tau' \subseteq t' \) be the subset of tasks in \( t' \) that are scheduled over \([t', t' + \delta] \). Because \( \epsilon \in (0, \delta] \), these tasks are also the tasks scheduled over \([t', t' + \epsilon] \). Thus,

\[
\text{Dev}(\tau', t' + \epsilon) = \text{Dev}(\tau', t' + \epsilon) + \text{Dev}(\tau' \setminus \tau', t' + \epsilon) \\
\leq \{ \text{By Lemmas 6 and 7} \} \\
\text{Dev}(\tau', \tau') + \epsilon(U(\tau') - |\tau|) + \text{Dev}(\tau' \setminus \tau', t' + \epsilon) \\
= \{ \text{By Lemma 10} \} \\
\epsilon(U(\tau') - M) + \text{Dev}(\tau', t') \\
\leq \{ \text{By Corollary 2} \} \\
\text{Dev}(\tau', t'). \\Box
\]

**Lemma 11.** For any time \( t \) and task set \( t' \subseteq \tau \), all \( \delta \in [0, t] : \text{Dev}(\tau', t - \delta) \geq \text{Dev}(\tau', t) - U(\tau')\delta \).

**Proof.**

\[
\text{Dev}(\tau', t) - U(\tau')\delta = \{ \text{By Def. 2} \} \\
\sum_{t' \in t'} u(t - \delta - v(t) - \tau') \\
= \{ \text{Def. of } U(\tau') \} \\
\sum_{t' \in t'} u(t - \delta - v(t) - \tau') \\
\leq \{ \text{By Lemma 4} \} \\
\sum_{t' \in t'} u(t - \delta - v(t) - \tau') \\
= \{ \text{By Def. 2} \} \\
\text{Dev}(\tau', t - \delta) \Box
\]

**Lemma 12.** Under AC, \( \forall \delta \geq 0, \text{Dev}(\tau', t) \leq \beta(\tau') \)

where \( \beta(\tau') \) is defined as

\[
\forall \tau' \subseteq \tau(t) : \beta(\tau') = (T_{\text{max}} + \frac{2mC_{\text{max}}}{u_{\text{min}}} \frac{U(\tau')}{2u_{\text{min}}} (2m - U(\tau')). \tag{6}
\]

**Proof.** As a shorthand, let \( K = (T_{\text{max}} + 2mC_{\text{max}}/u_{\text{min}})/(2u_{\text{min}}) \) \( > 0 \). By Def. 2 and (6), (5) is true at \( t = 0 \). Assuming (5) is falsified, let \( t_{b} \) (for “boundary”) be the last time instant such that (5) is true over \([0, t_{b}] \). We prove the lemma by contradicting the definition of \( t_{b} \).

**Claim 12.1.** \( t_{b} \geq C_{\text{max}} \)

**Proof.** We prove by contradiction. Assume to the contrary that \( t_{b} < C_{\text{max}} \). Then \( \exists t \in [t_{b}, C_{\text{max}}) \) such that for some task set \( \tau' \subseteq \tau(t) \), \( \text{Dev}(\tau', t) > \beta(\tau') \).

\[
0 = \{ \text{By Def. 2} \} \\
\text{Dev}(\tau', t) = \text{Dev}(\tau', t - t) \\
\geq \{ \text{By Lemma 11} \} \\
\text{Dev}(\tau', t) - U(\tau')t \\
\geq \{ \text{Dev}(\tau', t) > \beta(\tau', t) \} \\
\beta(\tau', t) - U(\tau')C_{\text{max}} \\
= \{ \text{By (6)} \} \\
KU(\tau')(2m - U(\tau')) \\
= \{ \text{by (1)} \} \\
U(\tau')mK - C_{\text{max}} \\
\geq \{ \text{m } \geq 1 \text{ and } K > mC_{\text{max}}/u_{\text{min}}^{2} \geq C_{\text{max}} \} \\
U(\tau')(C_{\text{max}} - C_{\text{max}}) \\
= 0. \Box
\]
0 > 0 is a contradiction, hence, \( t_b \geq C_{\text{max}} \).

**Claim 12.2.** At time \( t_b \),

\[
\forall r' \subseteq r(t_b) : \text{Dev}(r', t_b) \leq \beta(r'),
\]
\[
\exists t^b \subseteq r(t_b) : \forall \delta > 0 : \exists e \in (0, \delta) : \text{Dev}(t^b, t_b + e) > \beta(t^b)
\]
\[
\wedge \text{Dev}(t^b, t_b) = \beta(t^b).
\]

**Proof.** (7) follows from the definition of \( t_b \).
If (8) is false, then \( \forall r^b \subseteq r : \exists \delta > 0 : \forall e \in (0, \delta) : \text{Dev}(t^b, t_b + e) \leq \beta(t^b) \). This means that (5) is true over \([t_b, t_b + \delta] \), which contradicts the definition of \( t_b \).
(9) follows from (7), (8), and Lemma 5.

Let \( r^b \) be any of the task subsets known to exist by (8).

**Claim 12.3.** For any \( \delta \in [0, C_{\text{max}}] \), for any task \( r_e \in r^b \),

\[
\text{Dev}(r_e, t_b - \delta) \geq K(2m - 2\muU(t^b) + u_e)u_e - U(t^b)\delta
\]

**Proof.**

\[
\text{Dev}(r_e, t_b - \delta) = \{\text{By Def. 2}\}
\]
\[
\text{Dev}(t^b, t_b - \delta) - \text{Dev}(t^b \setminus \{r_e\}, t_b - \delta)
\]
\[
\geq \{\text{By Lemma 11}\}
\]
\[
\text{Dev}(t^b, t_b) - U(t^b)\delta - \text{Dev}(t^b \setminus \{r_e\}, t_b - \delta)
\]
\[
\geq \{\text{By Def. of } t_b\}
\]
\[
\text{Dev}(t^b, t_b) - U(t^b)\delta = \beta(t^b \setminus \{r_e\})
\]
\[
\geq \{\text{By (9)}\}
\]
\[
\beta(t^b) - \beta(t^b \setminus \{r_e\}) - U(t^b)\delta
\]
\[
= \{\text{By (6)}\}
\]
\[
KU(t^b)(2m - U(t^b)) - KU(t^b \setminus \{r_e\})(2m - U(t^b \setminus \{r_e\})) = U(t^b)\delta
\]
\[
K(2m - 2\muU(t^b) + u_e)u_e - U(t^b)\delta
\]

**Claim 12.4.** Any task \( r_e \in r^b \) has a ready job over \([t_b - C_{\text{max}}, t_b] \).

**Claim 12.5.** For any \( \delta \in [0, C_{\text{max}}] \), for any task \( r_e \in r(t_b - \delta) \setminus r^b \),

\[
\text{Dev}(r_e, t_b - \delta) \leq K(2m - 2\muU(t^b) - u_e)u_e + U(t^b)\delta
\]

**Proof.**

\[
\text{Dev}(r_e, t_b - \delta) = \{\text{By Def. 2}\}
\]
\[
\text{Dev}(t^b \cup \{r_e\}, t_b - \delta) - \text{Dev}(t^b, t_b - \delta)
\]
\[
\leq \{\text{By Lemma 11}\}
\]
\[
\text{Dev}(t^b \cup \{r_e\}, t_b - \delta) - \text{Dev}(t^b, t_b) + U(t^b)\delta
\]
\[
\leq \{\text{By Def. of } t_b\}
\]
\[
\beta(t^b \cup \{r_e\}) - \beta(t^b) + U(t^b)\delta
\]
\[
= \{\text{By (6)}\}
\]
\[
KU(t^b \cup \{r_e\})(2m - U(t^b \cup \{r_e\})) = U(t^b)\delta
\]
\[
K(2m - 2\muU(t^b) - u_e)u_e + U(t^b)\delta
\]

**Claim 12.6.** For any time \( t \in [t_b - C_{\text{max}}, t_b] \), for any tasks \( r_e \in r^b \) and \( r_e \in r(t) \setminus r^b \), if \( r_e \) has a ready job at \( t \), then \( d_e(t) < d_t(t) \).
Proof. Let $\delta = t_b - t$. Then $\delta \in [0, C_{\text{max}}]$.

\[
\frac{\text{Dev}(\tau_e, t)}{u_e} - \frac{\text{Dev}(\tau_i, t)}{u\ell} = \frac{\text{Dev}(\tau_e, t_b - \delta)}{u_e} - \frac{\text{Dev}(\tau_i, t_b - \delta)}{u\ell} \quad (\delta = t_b - t)
\]

\[
\geq \{\text{By Claims 12.3 and 12.5}\}
\]

\[
K(u_e + u\ell) - U(t^b)\delta \left( \frac{1}{u_e} + \frac{1}{u\ell} \right)
\]

\[
\geq m \geq U(t^i) \text{ by (1) and } \delta \leq C_{\text{max}}
\]

\[
K(u_e + u\ell) - mC_{\text{max}} \left( \frac{1}{u_e} + \frac{1}{u\ell} \right)
\]

\[
\geq K(2u_{\text{min}}) - 2mC_{\text{max}}/u_{\text{min}}
\]

\[
\geq T_{\text{max}}
\]

The claim follows from Claim 12.4 and Lemma 3. ▶

By Claim 12.6, Corollary 3, and (9), there exists $\delta > 0$ such that for all $\epsilon \in (0, \delta]$, $\text{Dev}(t^b, t_b + \epsilon) \leq \text{Dev}(t^b, t_b) = \beta(t^b)$. This contradicts (8). Thus, $t_b$ does not exist, implying that (5) is true over $[0, \infty)$.

\[\Box\]

**Theorem 1.** Under our modified AC, the tardiness of any task $\tau_i$ is at most

\[
T_{\text{max}} + \frac{2mC_{\text{max}}}{u_{\text{min}}} \quad \left( \frac{2m - u_i}{2u_{\text{min}}} \right)
\]

(10)

Proof. By Lemma 12, $\text{Dev}(\tau_i, t) \leq \left( T_{\text{max}} + \frac{2mC_{\text{max}}}{u_{\text{min}}} \right) \frac{u_i}{2u_{\text{min}}} (2m - u_i)$ for all $t \geq 0$. By Corollary 1, the bound in (10) follows. □