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:
Stephen Tang, James H. Anderson, and Luca Abeni. 2021. On the Defectiveness of SCHED_DEADLINE w.r.t. Tardiness and Affinities, and a Partial Fix. In *29th International Conference on Real-Time Networks and Systems (RTNS'2021), April 7–9, 2021, NANTES, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3453417.3453440

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).

RTNS'2021, April 7–9, 2021, NANTES, France
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9001-9/21/04…$15.00
https://doi.org/10.1145/3453417.3453440

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. AC does not guarantee bounded tardiness under DL, even under an idealized abstraction of the scheduler. This is because affinities are far from 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, available online [14]). 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 τ_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 τ_i over all its jobs. We let T_i denote the period of task τ_i. The utilization u_i of task τ_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 τ_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 τ_i is the supremum of the tardiness of its jobs. If the tardiness of all tasks in τ is bounded under a given scheduler, then τ is *soft real-time (SRT)-schedulable* under that scheduler. τ 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.

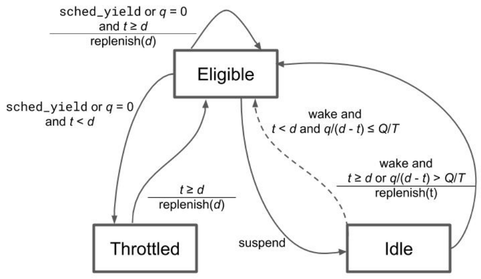
At any time, a task is either *active* or *inactive*.[1] A task is initially inactive, and must become active in order to release jobs. Each task becomes active at most once, and after becoming inactive, remains so indefinitely (this is for simplicity of notation, as a task becoming active a second time 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 π_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

[1]Note that "active" and "inactive" in this work are distinct from their definitions in DL's documentation and code.

Figure 1: CBS State Diagram


sched_yield or $q = 0$ and $t \geq d$
replenish(d)
Eligible
sched_yield or $q = 0$ and $t < d$
wake and $t < d$ and $q/(d - t) \leq Q/T$
wake and $t \geq d$ or $q/(d - t) > Q/T$
replenish(t)
$t \geq d$
replenish(d)
Throttled
suspend
Idle

analogous to a task's WCET, period, and utilization, respectively. A CBS's current budget and deadline are denoted q and d, respectively. In this subsection, we discuss how the CBS mimics and differs from the sporadic task model. Note that while we apply a notion of jobs to the CBS 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.[2] 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]The behavior of a CBS differs slightly if deadlines are not implicit.

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 *cpuset* is Linux's notion of a cluster[3]). 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,

$$\forall t : \sum_{\tau_i \in \tau(t)} u_i \leq \frac{\texttt{sched\_rt\_runtime\_us}}{\texttt{sched\_rt\_period\_us}} |\pi(t)|, \qquad (1)$$

where `sched_rt_runtime_us` and `sched_rt_period_us`[4] are set with the `proc` virtual file system and have default values of 950000 and 1000000, 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.

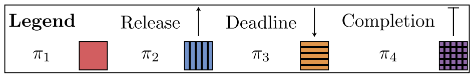
2.4 Fine-Grained Affinities

In Linux, the `sched_setaffinity` system call is an auxiliary method to the `cpuset` subsystem of setting affinities. `sched_setaffinity` defers 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.

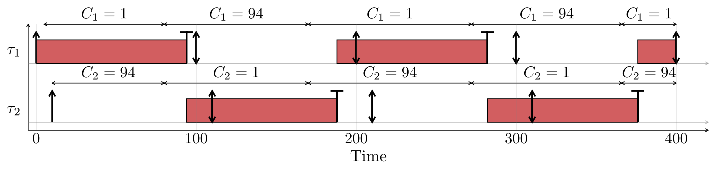
The usage of `sched_setaffinity` is not permitted for `DL` tasks unless AC is disabled by writing -1 to `sched_rt_runtime_us`. AC must be disabled as bounded tardiness to tasks is not guaranteed under arbitrary affinities.

[3]Though the `cpuset` subsystem actually organizes cpusets hierarchically, `DL` requires that cpusets with `DL` tasks to be exclusive from each other, which we also assume.
[4]These file names derive from `SCHED_RT`, whose accounting code `DL` intertwines with. We assume no RT tasks to focus on `DL`.

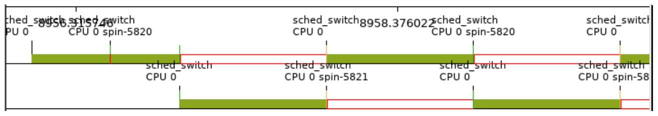
Figure 2: Unbounded tardiness due to dynamic tasks. (Legend is reused throughout paper.)



(a) Request pattern with unbounded tardiness.



(b) Corresponding trace.



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 practicable, we present scheduling traces of real DL tasks produced by `trace-cmd` and visualized with `kernelshark` 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 remaining budget and deadline are unchanged. 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 π_1 and begins with no tasks. At time $t = 0$, task τ_1 requests to enter DL with $(C_1, T_1) = (94, 100)$. This task is accepted because doing so will not violate (1). τ_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 τ_2 requests to enter with $(C_2, T_2) = (94, 100)$. This request is accepted by AC because τ_1 reduced its utilization. τ_2 releases its first job with $C_{2,1} = 94$ and $d_{2,1} = 110$. After this point, both tasks release jobs periodically.

However, prior to when τ_1 returns from the throttled state (see Fig. 1), τ_2 changes C_2 to 1 and τ_1 changes C_1 to 94. When the next job of τ_1 becomes ready at $t = 100$, this job's budget and deadline are determined entirely by τ_1's parameters at the instant it becomes ready. Thus, the budget of the next job is set to 94. Likewise, prior to when τ_2 is replenished at $t = 188$ (recall from Fig. 1 that a tardy task is replenished once its prior job completes), τ_1 sets C_1 to 1 such that τ_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 τ_1 and τ_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 depleting 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 $U_{\text{act}}(t)$. If $U_{\text{act}}(t) < 1$, then the frequency of the corresponding CPU is scaled by $U_{\text{act}}(t)$ of its maximum frequency and the budget of any task executing on said CPU is depleted at rate $U_{\text{act}}(t)$ (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 $U_{\text{act}} \leq 0.63$. 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 cap(π_i) denote the capacity of CPU π_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 deadline after the migration. As far as we are aware, it is not proven whether or not such migrations can result in unbounded tardiness, though we conjecture that it does not.

Even if such migrations between cpusets do not result in tardiness becoming unbounded, such migrations can cause higher tardiness than possible when tasks' cpusets are static.

▹ **Ex. 3.** (Corresponds with Fig. 3.) Consider two cpusets, one with a single CPU π_1 and a task τ_5 with $(C_5, T_5) = (2, 10)$ and another with three CPUs π_2, π_3, and π_4 and four tasks τ_1, τ_2, τ_3, and τ_4 all with $(C, T) = (70, 100)$.

At time $t = 92$, τ_4 requests to migrate to the cpuset containing π_1 and τ_5. This is accepted by AC as τ_4 and τ_5's combined utilization is less than 0.95. However, τ_4 keeps its current deadline and budget when it migrates, thereby causing τ_5 to become tardy when it otherwise would not have been. ◃

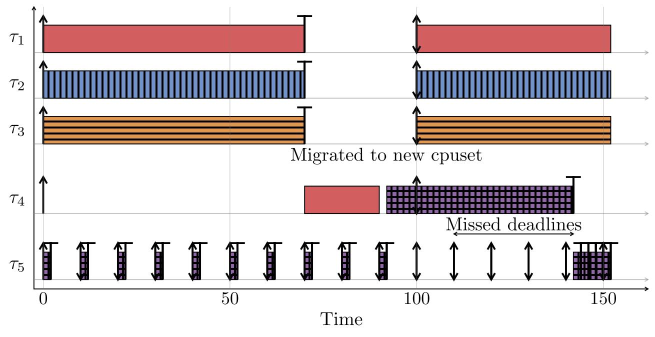
Obs. 1. Inter-cpuset migration can cause higher tardiness than expected under DL.

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-suspensions are known to cause capacity loss [12].

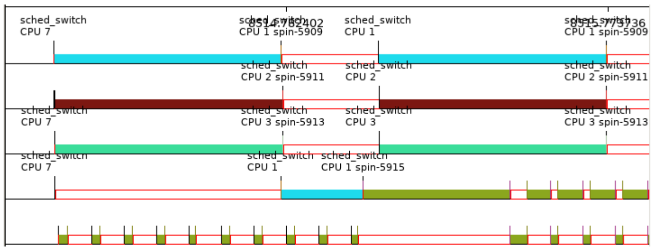
[5]The actual value assigned is 1024. We discuss relative to 1.0 for simplicity.

Figure 3: Migration between cpusets.

(a) Request pattern that causes additional tardiness.



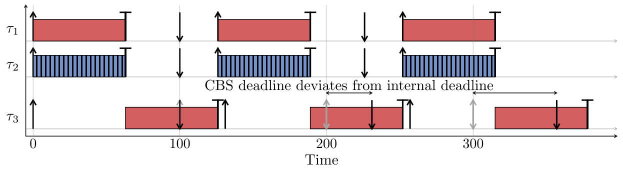
(b) Corresponding trace.



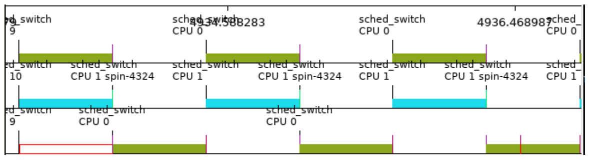
sched_switch CPU 7
sched_switch CPU 1 spin-5909
sched_switch CPU 1
sched_switch CPU 2 spin-5911
sched_switch CPU 2
sched_switch CPU 3 spin-5913
sched_switch CPU 3
sched_switch CPU 1 spin-5915

Figure 4: Suspensions inflate periods.

(a) Release pattern with self-suspending jobs.



(b) Corresponding trace.



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 π_1 and π_2 and three servers τ_1, τ_2, and τ_3 all with $(C, T) = (63, 100)$. However, the encapsulated real-time task of τ_3 must suspend briefly at the end of every job, perhaps to do some I/O. Meanwhile, τ_1 and τ_2 only suspend between jobs when none are available to execute.

All servers release jobs at $t = 0$, with τ_1 and τ_2 winning the deadline tie. This causes τ_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 τ_3 to become non-periodic, even if the encapsulated workload is periodic. For example, repeating the release pattern in Fig. 4a causes τ_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-suspensions result in capacity loss for sporadic tasks, while we conjecture 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 from the current implementation. Last, we explain how our patch modifies the implementation. Our patch is available online [14].

4.1 Bypassing the Throttled State

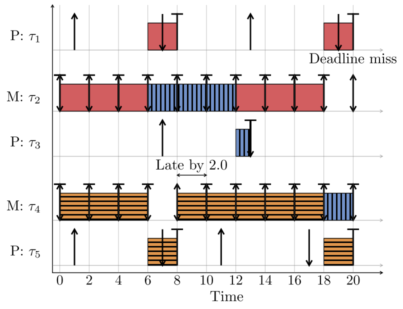
Problem. In DL, a task that completes a job by exhausting its budget or calling sched_yield after its deadline will remain eligible, bypassing the throttled state (as in Fig. 1). If so, this task continues executing its next job on the same CPU if it is not preempted by a different task with an earlier 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][6]; corresponds with Fig. 5.) Consider a cpuset with CPUs π_1, π_2, and π_3 with tasks τ_1, τ_2, τ_3, τ_4, and τ_5. Let $(C_1, T_1) = (C_5, T_5) = (2, 6)$, $(C_2, T_2) = (C_4, T_4) = (2, 2)$, and $(C_3, T_3) = (1, 6)$. τ_1, τ_3, and τ_5 are partitioned on π_1, π_2, and π_3, respectively, while τ_2 and τ_4 are migrating.

τ_2 and τ_4 release jobs periodically. Initially, τ_2 and τ_4 execute on π_1 and π_3, respectively. At $t = 6$, fixed tasks τ_1 and τ_5 preempt τ_2 and τ_4, respectively. The only other processor available to both τ_2 and τ_4 is π_2 (τ_2 cannot preempt τ_5 on π_3 and τ_4 cannot preempt τ_1 on π_1), which they cannot both use. We assume the tiebreak here favors τ_2 and it is scheduled, while τ_4 does not execute until $t = 8$ when it resumes execution on π_3. τ_2 is also forced to migrate off of π_2 by fixed task τ_3 at $t = 12$. This repeats at $t = 18$, except here τ_4 is scheduled over τ_2 because it is tardy by 2.0 time units due to not being scheduled over $[6, 8]$. As a result, τ_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 τ_2 or τ_4 increases by 2.0. ◁

[6] Ex. 4 of [15] actually does not consider a SP system, as τ_2 and τ_4 were not allowed to migrate to π_3 and π_1, respectively in [15]. Observe when considering Ex. 5 that the same schedule occurs even when such migrations are allowed given specific tie-breaking assumptions.

Figure 5: (Fig. 2(b) of [15]) Unbounded tardiness under DL. (P for partitioned and M for migrating.)



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.

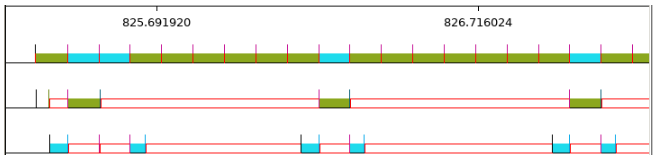
Let the deadline of a CPU be the deadline of the eligible task with earliest deadline on its dl_rq. The target CPU of a push is either a CPU with no DL tasks on its dl_rq (called a *free CPU*) or, if the cpuset has no free CPUs, the CPU with the latest deadline among the other CPUs in the cpuset. Note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

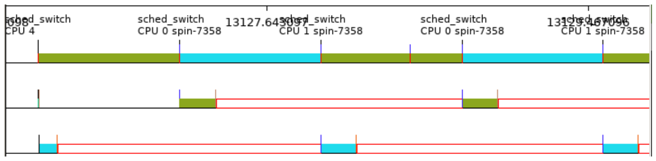
A task that bypasses the throttled state is immediately enqueued back onto its CPU's dl_rq with replenished budget and updated deadline. If another task on this CPU's dl_rq now has an early enough deadline to preempt the replenished task, it does so. Otherwise, the replenished task continues executing on the same CPU.

Figure 6: Differences in throttling behavior.

(a) Tasks do not bypass throttled state in our patch.



(b) Task bypasses throttle in original implementation.



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, τ_2 does not migrate to free CPU π_2 even though its current CPU π_1 has a partitioned task τ_1 with a ready job. τ_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 `dl_task_timer`, an `hrtimer` callback), it is enqueued back onto this CPU's `dl_rq` with a replenished budget and updated deadline. Because the replenished task became eligible, this CPU attempts to push a thread.

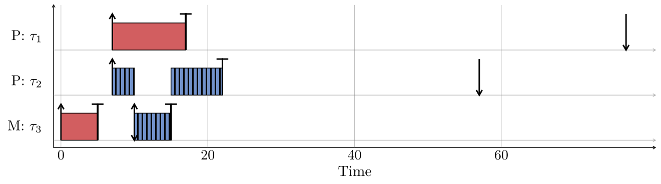
Patch. Ex. 5 would not have unbounded tardiness if successive jobs of tasks τ_2 and τ_4 would migrate when a free CPU or CPU with later deadline is available, thereby allowing partitioned tasks to execute. For example, if τ_2 had migrated to free CPU π_2 at time 2 instead of 6 in Fig. 5, then the first job of τ_1 would have been able to execute on π_1 over the interval $[2, 4]$. This would have freed π_1 over $[6, 8]$, allowing migrating task τ_4 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, τ_2 completes its job. Under our patch, τ_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, τ_2 would be pushed from π_1 to free CPU π_2. This is the schedule described in the previous paragraph that reduces tardiness.

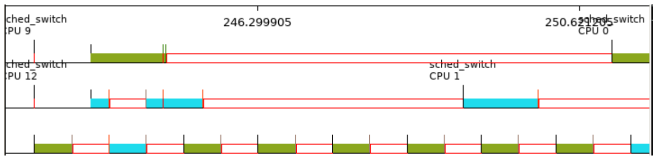
Note that callback function `dl_task_timer` will not push a scheduled task, as it is not safe to migrate an executing task. This is problematic because throttled tasks may still be scheduled because rescheduling is not instantaneous. To guarantee that a tardy task is pushed, `dl_task_timer` must wait for the tardy task to be unscheduled. Unfortunately, it does not help to wait within `dl_task_timer`

Figure 7: Pushes can cause priority inversions.

(a) Release pattern that causes priority inversion.



(b) Corresponding trace.



for the relevant CPU to reschedule, as both `dl_task_timer` and the rescheduling code both must acquire this CPU's `rq` lock (the `rq` struct contains generic runqueue fields as well as scheduler-specific data structures such as the `dl_rq`). In our patch, when `dl_task_timer` executes and observes that the task to be pushed is still scheduled, the callback releases the `rq` lock and retries in the future, giving the relevant CPU the chance to reschedule. This is done with `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 π_1 and π_2 and three tasks τ_1, τ_2, and τ_3 with $(C_1, T_1) = (10, 70)$, $(C_2, T_2) = (10, 50)$, and $(C_3, T_3) = (5, 10)$. τ_1 and τ_2 are partitioned on π_1 and π_2, respectively, while τ_3 is migrating. τ_3 releases its first job at $t = 0$ and executes on CPU π_1 until $t = 5$, at which point τ_3 is throttled. At time $t = 7$, both τ_1 and τ_2 release their first jobs and begin executing. At time $t = 10$, τ_3 returns from the throttled state (recall Fig. 1). This results in τ_3 being placed back onto π_1's `dl_rq` (as π_1 was τ_3's last CPU) and π_1 attempting to push τ_3.

Because, at the instant τ_3 is pushed, π_1 executes τ_1 with deadline 77 and π_2 executes τ_2 with deadline 57, τ_3 should remain on π_1 and preempt τ_1, whose deadline is later than that of τ_2. The actual behavior exhibited by DL is that π_1 will push τ_3 to π_2, preempting τ_2. This is confirmed in Fig. 7b. ◁

Implementation. The search for the target CPU is facilitated by a per-cpuset `cpudl` struct. A `cpudl` 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 `dl_rq`, respectively.

In order to be pushed to a target CPU, a task must be on the pushing CPU's `dl_rq`. Because this task was enqueued onto this `dl_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 `dl_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_dl_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 `dl_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`). Dequeueing 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 `dl_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 `dl_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 `dl_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 $\tau(t)$, changing the CPUs in $\pi(t)$, or modifying either of `sched_rt_runtime_us` or `sched_rt_period_us`.

Unfortunately, there is no common helper function used to check (1) for all of these events. For adding tasks to $\tau(t)$, (1) is checked in function `__dl_overflow` (this is also where (1) is checked when task parameters are changed or a task from another cpuset is added to this cpuset, but we assume these events do not occur for the reasons discussed in Sec. 3). For changing the CPUs in $\pi(t)$, (1) is checked in function `dl_cpuset_cpumask_can_shrink`. For modifying `sched_rt_runtime_us` or `sched_rt_period_us`, (1) is checked in function `sched_dl_global_validate`.

Adding a task to $\tau(t)$ using `sched_setattr` will fail if the server to add does not have affinity for every CPU in $\pi(t)$. 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,

$$\forall t : \forall \pi_j \in \pi(t) : \sum_{\tau_i \in \tau(t, \pi_j)} u_i \leq \frac{\texttt{sched\_rt\_runtime\_us}}{\texttt{sched\_rt\_period\_us}}. \quad (2)$$

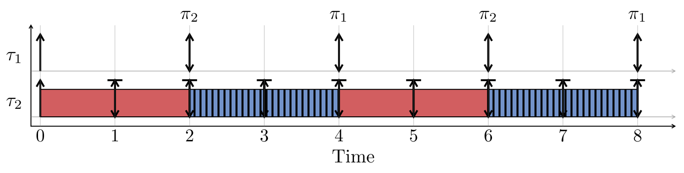
This is done by adding checks for (2) where `__dl_overflow` is called and in `sched_dl_global_validate`, which can cause (2) to be violated by adding a new task to $\tau(t, \pi_j)$ or modifying its r.h.s. Similar to how the l.h.s. of (1) is tracked in `total_bw` stored in the `root_domain`, the l.h.s.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

(2) is not checked in `dl_cpuset_cpumask_can_shrink`, which is called when $\pi(t)$ 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 $\pi(t)$.

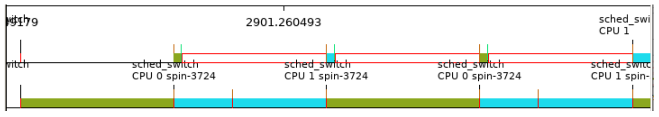
4.4 Dynamic Fine-Grained Affinities

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.

Figure 8: Dynamic affinities can starve tasks.



(a) Release pattern with dynamic affinities.


2901.260493
sched_switch CPU 0 spin-3724
sched_switch CPU 1 spin-3724
sched_switch CPU 0 spin-3724
CPU 1

(b) Corresponding trace.

▹ **Ex. 7.** This example corresponds with Fig. 8. Consider a SP system on a cpuset with two CPUs π_1 and π_2 and two tasks τ_1 (initially partitioned on π_1) and τ_2 migrating with $(C_1, T_1) = (1, 2)$ and $(C_2, T_2) = (1, 1)$. Both tasks release jobs at time $t = 0$.

τ_2, whose deadline is earlier than τ_1's deadline, executes on π_1 until preempted by τ_1 at $t = 2$. τ_2 migrates to π_2 once preempted. However, τ_1 requests to change its affinity from being partitioned on π_1 to π_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), τ_1's deadline is updated from 2 to 4 when it migrates at $t = 2$. Thus, τ_1 does not have an early enough deadline to preempt τ_2, and so it continues to be unscheduled.

Repeating this pattern of dynamic affinity requests can prevent τ_1 from executing indefinitely, as in Fig. 8a (the reason that the workload corresponding to τ_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, `sched_setaffinity` performs the same check between the cpuset's CPUs and the requested affinity as `__sched_setscheduler`. Thus, effectively `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 `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 `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 *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 *transient*.

▹ **Ex. 8.** Consider a time instant such that an ARPI exists between partitioned task τ_1, migrating task τ_2 executing on τ_1's CPU, and task τ_3 with later deadline than τ_1.

Under our patched `DL`, this priority inversion lasts for at most the execution time of τ_2's job. When τ_2's job completes, τ_2 will always be throttled (recall from Sec. 4 that all tasks that complete jobs are throttled in our patch), and will migrate to τ_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 τ_2 to be preempted before its job completes for the duration of the ARPI, as any migrating task will preempt τ_3 before τ_2 under our patched `DL` because τ_3 has a later deadline. If a partitioned task preempts τ_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 τ_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 online [14].

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 `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 `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 `dl_task_timer` returns said task to a runqueue. To measure the duration of pushes, we also inserted tracepoints around `push_dl_task`. To get a more holistic view of how these changes to

Figure 9: Forced Throttle Duration

(a) 16 Tasks (b) 40 Tasks

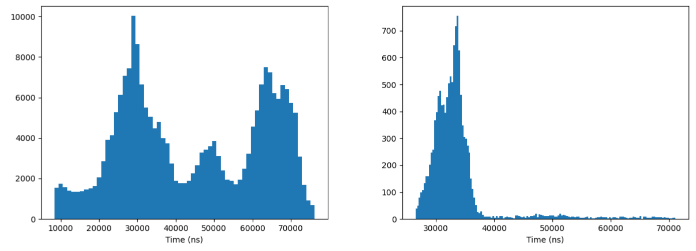
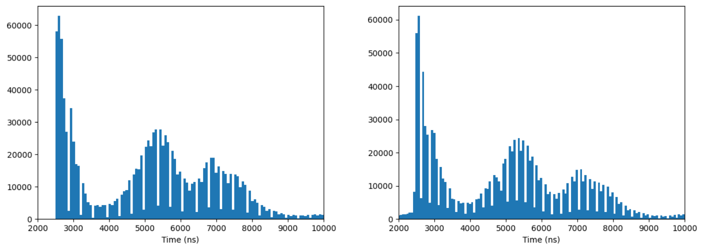


Figure 10: Push Durations (40 Tasks)

(a) Original (b) Patched



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.

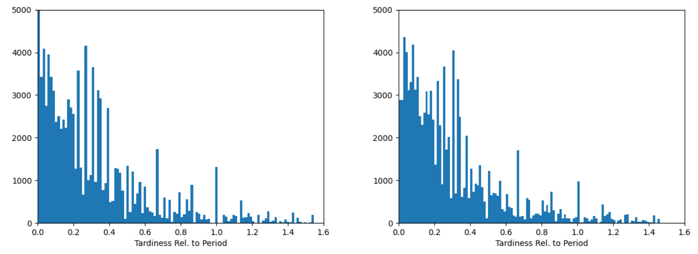
Latency of forced throttles. Note that we do not compare against the original DL implementation when considering forced task throttling because this overhead is unique to our variant. The distribution of sampled durations during which our patched DL forced a task to be throttled when it would not have 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 hr_timer_forward_now with the dl_task_timer callback to ensure that tasks forced into the throttle state are unscheduled before dl_task_timer attempts to push a task. The time difference between peaks roughly correlates with the time interval the hrtimer is forwarded by in our patched DL. As each usage of hr_timer_forward_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 hrtimer as we suspect, an alternative method for forcing tasks that complete jobs to migrate that avoids using the hrtimer 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 push_dl_task contains a retry loop in case the state of the system changes while rq locks are dropped in double_lock_balance. The effect of the added enqueue

Figure 11: Tardiness (40 Tasks)

(a) Original (b) Patched



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 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.

ACKNOWLEDGMENTS

Work was supported by NSF grants CNS 1563845, CNS 1717589, CPS 1837337, CPS 2038855, and CPS 2038960, ARO grant W911NF-20-1-0237, and ONR grant N00014-20-1-2698.

REFERENCES

[1] 2009. rt-app. https://github.com/scheduler-tools/rt-app. Online; accessed 23 Oct 2020.

[2] 2018. Deadline Task Scheduling. https://github.com/torvalds/linux/blob/master/Documentation/scheduler/sched-deadline.rst. Online; accessed 03 June 2020.

[3] 2019. SCHED_DEADLINE on heterogeneous multicores. https://lwn.net/Articles/793495/. Online; accessed 15 Oct 2020.

[4] L. Abeni, G. Lipari, A. Parri, and Y. Sun. 2016. Multicore CPU reclaiming: parallel or sequential? 1877–1884. https://doi.org/10.1145/2851613.2851743

[5] F. Cerqueira, A. Gujarati, and B. B. Brandenburg. 2014. Linux's Processor Affinity API, Refined: Shifting Real-Time Tasks Towards Higher Schedulability. In *2014 IEEE Real-Time Systems Symposium*. 249–259. https://doi.org/10.1109/RTSS.2014.29

[6] U. M. C. Devi and J. H. Anderson. 2005. Tardiness bounds under global EDF scheduling on a multiprocessor. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*. 12 pp.–341.

[7] P. Emberson, R. Stafford, and R.I. Davis. 2010. Techniques For The Synthesis Of Multiprocessor Tasksets. *WATERS'10* (01 2010).

[8] A. Gujarati, F. Cerqueira, and B. B. Brandenburg. 2013. Outstanding Paper Award: Schedulability Analysis of the Linux Push and Pull Scheduler with Arbitrary Processor Affinities. In *2013 25th Euromicro Conference on Real-Time Systems*. 69–79.

[9] A. Gujarati, F. Cerqueira, and B. B. Brandenburg. 2014. Multiprocessor real-time scheduling with arbitrary processor affinities: from practice to theory. *Real-Time Systems* 51 (2014), 440–483.

[10] J. Lelli. 2014. taskgen. https://github.com/jlelli/taskgen. Online; accessed 23 Oct 2020.

[11] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli. 2016. Deadline Scheduling in the Linux Kernel. *Softw. Pract. Exper.* 46, 6 (June 2016), 821–839. https://doi.org/10.1002/spe.2335

[12] C. Liu and J. H. Anderson. 2012. An O(m) Analysis Technique for Supporting Real-Time Self-Suspending Task Systems. In *2012 IEEE 33rd Real-Time Systems Symposium*. 373–382. https://doi.org/10.1109/RTSS.2012.87

[13] C. Scordino, L. Abeni, and J. Lelli. 2018. Energy-Aware Real-Time Scheduling in the Linux Kernel. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing* (Pau, France) *(SAC '18)*. Association for Computing Machinery, New York, NY, USA, 601–608. https://doi.org/10.1145/3167132.3167198

[14] S. Tang and J. H. Anderson. 2020. On the Defectiveness of SCHED_DEADLINE w.r.t. Tardiness and Affinities, and a Partial Fix. Full version of this paper, available at http://jamesanderson.web.unc.edu/papers/.

[15] S. Tang and J. H. Anderson. 2020. Towards Practical Multiprocessor EDF with Affinities. In *41st IEEE Real-Time Systems Symposium*.

[16] S. Tang, S. Voronov, and J. H. Anderson. 2019. GEDF Tardiness: Open Problems Involving Uniform Multiprocessors and Affinity Masks Resolved. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 133)*, Sophie Quinton (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 13:1–13:21. https://doi.org/10.4230/LIPIcs.ECRTS.2019.13

[17] S. Voronov and J. H. Anderson. 2018. An Optimal Semi-Partitioned Scheduler Assuming Arbitrary Affinity Masks. In *2018 IEEE Real-Time Systems Symposium (RTSS)*. 408–420.

[18] K. Yang and J. Anderson. 2017. On the Soft Real-Time Optimality of Global EDF on Uniform Multiprocessors. In *2017 IEEE Real-Time Systems Symposium (RTSS)*. 319–330. https://doi.org/10.1109/RTSS.2017.00037