Extending Soft Real-Time Analysis for Heterogeneous Multiprocessors

Stephen Tang

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2024

Approved by:

James H. Anderson

F. Donelson Smith

Samarjit Chakraborty

Luca Abeni

Sanjoy Baruah

ABSTRACT

Stephen Tang: Extending Soft Real-Time Analysis for Heterogeneous Multiprocessors
(Under the direction of James H. Anderson)

Though prior work has established the Soft Real-Time (SRT)-optimality of Earliest-Deadline-First (EDF) variants on multiprocessors with identical and uniform speeds, these multiprocessor models preclude features such as processor affinities and are insufficient to describe modern multiprocessors, which have grown increasingly heterogeneous. This has had practical consequences, such as the inability of Linux's `SCHED_DEADLINE` admission-control system to make SRT guarantees in the presence of processor affinities or on heterogeneous machines. This fact highlights the need to extend theoretical results on EDF to more general multiprocessor models.

This dissertation fulfills this need by extending existing SRT analysis of window-constrained (WC) schedulers (of which EDF is a member) to heterogeneous multiprocessors. We derive improved response-time bounds for WC schedulers under uniform multiprocessors. We prove that a WC extension of an EDF variant called Strong-APA-EDF is SRT-optimal under arbitrary affinities (so long as processor speeds are identical). We define a WC scheduler variant targeting unrelated multiprocessors and derive response-time bounds. Additionally, we present patches for `SCHED_DEADLINE` with the purpose of restoring SRT guarantees for certain special cases of heterogeneous multiprocessors. These special cases are selected such that the increase in overheads due to our patches is manageable.

ACKNOWLEDGEMENTS

This dissertation is the result of the combined support of many people. This dissertation would never have been completed without the guidance of my adviser, Jim Anderson. Jim has fundamentally shaped my approaches to research, technical writing, presenting, and teaching. I would have given up on this dissertation countless times if not for Jim's continued encouragement that there is something worthwhile written here. In taking me on as a student, Jim has given me the opportunity to study a problem whose solution I feel is somehow 'fundamental.' Though I ultimately failed to close this problem, I am grateful that I got to experience the feeling of brushing up against a fundamental solution.

I am grateful for my committee members, Don Smith, Samarjit Chakraborty, Luca Abeni, and Sanjoy Baruah. I am grateful for your patience in serving for much longer than I originally expected. I am grateful for your diligent feedback on this document, which is much longer than I originally intended.

I thank the current and former students with whom I have been lucky to be a coauthor: Namhoon Kim, Micaiah Chisholm, Sergey Voronov, Nathan Otterness, Tanya Amert, Joshua Bakita, Shareef Ahmed, Sims Hill Osbourne, and Jingyuan Chen. Namhoon and Micaiah mentored me when I knew nothing as a fresh graduate student. I still don't know anything, but the effort is appreciated. I was fortunate to share both an office and an apartment with Sergey. Outside of being the student who I have collaborated with the most, I am infinitely grateful to you for convincing me to adopt Cola the cat.

I owe much to current and former UNC staff and faculty, especially Jodie Gregoritsch, Denise E. Kenney, Missy Wood, Jasleen Kaur, Don Porter, and the late Bil Hays.

I am grateful to my board game group of Tommy, Sam, Andy, and Josh. These sessions were welcome distractions during the times when research was frustrating. Please finally finish your Baldur's Gate campaign.

Words are insufficient to express the support I have received from my family. To my nephew, Alex, consulting with you over the phone has been invaluable for my research. I look forward to the discoveries you will surely make in the future.

Funding for this research was provided by NSF grants CNS 1409175, CNS 1563845, CNS 1717589, CPS 1446631, CPS 1837337, CPS 2038855, and CPS 2038960, AFOSR grant FA9550-14-1-0161, ARO

grants W911NF-14-1-0499, W911NF-17-1-0294, and W911NF-20-1-0237, ONR grant N00014-20-1-2698, and funding from General Motors.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

EDF	Earliest-Deadline-First
HRT	Hard Real-Time
SRT	Soft Real-Time
JLFP	Job-Level Fixed-Priority
JLDP	Job-Level Dynamic-Priority
WC	Window-Constrained
DVFS	Dynamic Frequency and Voltage Scaling
SIMD	Single Instruction Multiple Data
GPU	Graphics Processing Unit
DSP	Digital Signal Processor
FPGA	Field-Programmable Gate Array
ACS	Admission Control System
CVA	Compliant Vector Analysis
RTOS	Real-Time Operating System
EEVDF	Earliest Eligible Virtual Deadline First
CFS	Completely Fair Scheduler
PID	Process ID
TGID	Thread Group ID
TID	Thread ID

LIST OF SYMBOLS[1]

Symbol	Range	Meaning
τ	N/A	Task set
τ_i	τ	i^{th} task
n	$\mathbb{N}$	Number of tasks $\lvert\tau\rvert$
π	N/A	Processor set
π_j	π	j^{th} processor
m	$\mathbb{N}$	Number of processors $\lvert\pi\rvert$
C_i	$\mathbb{R}_{>0}$	Worst-case execution time of τ_i
T_i	$\mathbb{R}_{>0}$	Period of τ_i
D_i	$\mathbb{R}_{>0}$	Relative deadline of τ_i
u_i	$\mathbb{R}_{>0}$	Utilization $\frac{C_i}{T_i}$ of τ_i
$[i]$	$\mathbb{N}$	Subscript of i^{th} largest value, *e.g.*, $u_{[1]} = \max_{\tau_i \in \tau} u_i$
$sp^{(j)}$	$\mathbb{R}_{\geq 0}$	Speed of π_j under UNIFORM
$sp^{i,j}$	$\mathbb{R}_{\geq 0}$	Speed of π_j when executing τ_i under UNRELATED
$\tau_{i,j}$	N/A	j^{th} job of τ_i
$c_{i,j}$	$(0, C_i]$	Execution cost of $\tau_{i,j}$
$a_{i,j}$	$\mathbb{R}$	Arrival time of $\tau_{i,j}$, $a_{i,j} + T_i \leq a_{i,j+1}$
$d_{i,j}$	$\mathbb{R}$	Absolute deadline $a_{i,j} + D_i$ of $\tau_{i,j}$
$\tilde{d}_{i,j}$	$\mathbb{R}$	Implicit deadline $a_{i,j} + T_i$ of $\tau_{i,j}$
$f_{i,j}$	$\mathbb{R}$	Completion time of $\tau_{i,j}$
$a_i(t)$	$\mathbb{R}$	Arrival time of current job of τ_i at t
$d_i(t)$	$\mathbb{R}$	Absolute deadline $a_i(t) + D_i$ of current job of τ_i at t
$\tilde{d}_i(t)$	$\mathbb{R}$	Implicit deadline $a_i(t) + T_i$ of current job of τ_i at t
$c_i(t)$	$(0, C_i]$	Total execution cost of current job of τ_i at t
$rem_i(t)$	$(0, c_i(t)]$	Remaining execution cost of current job of τ_i at t
$rdy_{i,j}$	$\mathbb{R}$	Ready time of $\tau_{i,j}$, $rdy_{i,j} \leq a_{i,j}$
$\tau_{\text{rdy}}(t)$	$\mathbb{P}(\tau)$	Subset of ready tasks at t

[1] $\mathbb{P}(\mathbb{S})$ denotes the power set of set $\mathbb{S}$.

$zlt_{i,j}$	$\mathbb{R}$	Zero-lag time $a_{i,j} + \frac{c_{i,j}}{u_i}$ of $\tau_{i,j}$
$\tau_{\text{act}}(t)$	$\mathbb{P}(\tau)$	Subset of active tasks at t, $\tau_{\text{rdy}}(t) \subseteq \tau_{\text{act}}(t)$
n_{act}	$\mathbb{N}$	Maximum number of active tasks, $n_{\text{act}} \geq \max_t \lvert\tau_{\text{act}}(t)\rvert$
t_i^{act}	$\mathbb{R}$	i^{th} activation time instant (Definition 2.14)
$U(\tau')$	$\mathbb{R}_{\geq 0}$	Total utilization of $\tau' \subseteq \tau$
U_{max}	$\mathbb{R}_{>0}$	Maximum total utilization of active tasks, $U_{\text{max}} \geq \max_t U(\tau_{\text{act}}(t))$
α_i	$\mathbb{P}(\pi)$	Affinity set of τ_i
$pp_{i,j}(t)$	$\mathbb{R}$	Priority point of $\tau_{i,j}$ at t
$pp_i(t)$	$\mathbb{R}$	Priority point of current job of τ_i at t
ϕ	$\mathbb{R}_{\geq 0}$	Priority window (Definition 2.20)
$csp_i(t)$	$\mathbb{R}_{\geq 0}$	Speed of processor executing τ_i at t
$\mathbb{M}$	$\mathbb{P}(\tau \times \pi)$	Matching on affinity graph between τ and π
$vt_i(t)$	$\mathbb{R}$	Virtual time of τ_i at t (Definition 3.1)
$dev_i(t)$	$\mathbb{R}_{\geq 0}$	Deviation of τ_i at t (Definition 3.2)
$\mathcal{HP}(\tau', t)$	N/A	High-priority predicate (Definition 3.3)
$\mathcal{HP}$-$\mathcal{LAG}$	N/A	See Definition 3.4
$\beta_{\tau'}$	$\mathbb{R}$	See Definition 3.5
$\Psi_i(t)$	$\mathbb{R}_{\geq 0}$	Profit function of τ_i (Definition 3.6)
Δ	N/A	Symmetric difference operator (Definition 3.7)
$s\ell$	$(0, 1)$	Slowdown factor (Definition 3.8)
sp^{max}	$\mathbb{R}_{>0}$	Maximum speed under UNRELATED
β_{Unr}	$\mathbb{R}$	See Definition 3.10
π^{big}	$\mathbb{P}(\pi)$	Subset of big CPUs
π^{LIT}	$\mathbb{P}(\pi)$	Subset of LITTLE CPUs
m^{big}	$\mathbb{N}$	Number of big CPUs
m^{LIT}	$\mathbb{N}$	Number of LITTLE CPUs
τ^{big}	$\mathbb{P}(\tau)$	Subset of τ with affinity for π^{big}
τ^{LIT}	$\mathbb{P}(\tau)$	Subset of τ with affinity for π^{LIT}
τ^{glob}	$\mathbb{P}(\tau)$	Subset of τ with affinity for $\pi = \pi^{\text{big}} \cup \pi^{\text{LIT}}$

$\tau^{\text{big}}_{\text{act}}(t)$	$\mathbb{P}\big(\tau^{\text{big}}\big)$	Subset of τ^{big} that are active at t
$\tau^{\text{LIT}}_{\text{act}}(t)$	$\mathbb{P}\big(\tau^{\text{LIT}}\big)$	Subset of τ^{LIT} that are active at t
$\tau^{\text{glob}}_{\text{act}}(t)$	$\mathbb{P}\big(\tau^{\text{glob}}\big)$	Subset of τ^{glob} that are active at t
sp^{L}	$(0, 1.0)$	Speed of each LITTLE CPU
τ^{idle}	N/A	Set of idle tasks (see Definition 5.3)
$\bar{\tau}$	N/A	Augmented task set $\tau \cup \tau^{\text{idle}}$
$\bar{\tau}_{\text{act}}(t)$	N/A	Augmented active task set $\tau_{\text{act}}(t) \cup \tau^{\text{idle}}$
$\bar{\tau}_{\text{rdy}}(t)$	N/A	Augmented ready task set $\tau_{\text{rdy}}(t) \cup \tau^{\text{idle}}$
$\bar{\mathbf{X}}$	$\mathbb{R}_{\geq 0}^{(n+m)\times m}$	Augmented configuration (see Definition 5.5)
$d_j^{\text{CPU}}\big(\bar{\mathbf{X}}, t\big)$	$\mathbb{R}$	Deadline of task matched with π_j in $\bar{\mathbf{X}}$ at t
$d^{\text{big}}\big(\bar{\mathbf{X}}, t\big)$	$\mathbb{R}$	Latest deadline of any big CPU (see Definition 5.7)
$d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big)$	$\mathbb{R}$	Latest deadline of any LITTLE CPU (see Definition 5.7)
$\bar{d}_i\big(\bar{\mathbf{X}}, t\big)$	$\mathbb{R}$	Weighted deadline of τ_i under $\bar{\mathbf{X}}$ at t (see Definition 5.8)
$\bar{\mathbf{X}}^{\text{USE}}(t)$	$\mathbb{R}_{\geq 0}^{(n+m)\times m}$	Augmented configuration selected by Ufm-SC-EDF at time t
$\bar{\mathbf{X}}^{\text{opt}}(t)$	$\mathbb{R}_{\geq 0}^{(n+m)\times m}$	See Definition 5.13
$\bar{\mathbb{M}}^{\text{USE}}(t)$	$\mathbb{P}(\bar{\tau} \times \pi)$	Matching corresponding with $\bar{\mathbf{X}}^{\text{USE}}(t)$
$\bar{\mathbb{M}}^{\text{opt}}(t)$	$\mathbb{P}(\bar{\tau} \times \pi)$	Matching corresponding with $\bar{\mathbf{X}}^{\text{opt}}(t)$
u_i^{big}	$\mathbb{R}_{\geq 0}$	See Definition 5.14
$\tau^{\text{G}}(\tau')$	$\mathbb{P}(\tau')$	Global tasks in $\tau' \subseteq \tau$ under SEMI-PARTITIONED
$\tau^{\text{P}}(\tau', \pi_j)$	$\mathbb{P}(\tau')$	Tasks partitioned on π_j in $\tau' \subseteq \tau$ under SEMI-PARTITIONED
$\pi^{\text{P}}(\tau')$	$\mathbb{P}(\pi)$	Processors with partitioned tasks in $\tau' \subseteq \tau$ under SEMI-PARTITIONED
$\beta^{\text{DL}}_{\tau'}$	$\mathbb{R}$	See Definition A.2

CHAPTER 1: INTRODUCTION

A system is *real time* if the timeliness of its computational results is as significant as their logical correctness. For example, consider an autonomous vehicle running computer vision applications. If an obstacle enters the vehicle's path, the computer vision application must complete within a bounded amount of time to avoid a crash. Unlike other computing systems, in which fast average-case response times are desirable, the correctness of real-time systems is concerned with bounding the worst case.

Real-time tasks. Real-time workloads are *recurrent*, meaning that programs are expected to respond to inputs repeatedly over a long duration of time (*e.g.*, an autonomous vehicle is expected to regularly process camera frames one after another). These programs are called *tasks*, with the i^{th} task in a system traditionally denoted as τ_i. Different invocations of a tasks are called *jobs*, with the j^{th} job of task τ_i being denoted $\tau_{i,j}$. Each job has an *arrival time*, which is generally when the job becomes executable. Tasks are generally assumed to be singled-threaded, meaning that jobs of the same task may not execute in parallel. Timeliness requirements are expressed by assigning each job a *deadline*, which typically is some fixed offset (called a *relative deadline*) after said job's arrival time.

The traditional task model considered in the real-time community is the *sporadic* task model. Under the sporadic task model, task τ_i is constrained by its *worst-case execution time* C_i, the maximum units of execution required by any of its jobs, and T_i, the minimum interarrival time between any two of its consecutive jobs. Task τ_i's *utilization*, a measure of the long-term fractional demand of task τ_i for a 1.0-speed processor, is then C_i/T_i. A common assumption under the sporadic task model is *implicit deadlines*. For an implicit-deadline task τ_i, its relative deadline is equal to T_i.

Timeliness requirements are traditionally *hard real time (HRT)*. In such systems, it is required that each job must *provably* complete by its deadline. This is in contrast to *soft real-time (SRT)* systems that relax this requirement. For example, consider a streaming application that is expected to deliver a consistent number of video frames per second. Each frame must be processed within a bounded amount of time to achieve this frame rate, but the occasional late or dropped frame does not significantly impact quality of service.

Note that there exist alternative definitions of SRT in the literature. The interpretation of SRT that we consider in this dissertation is that each job has *bounded response time*, *i.e.*, the difference between each job's completion and arrival times must be *provably* bounded by some (ideally small) constant.

The ability for a system to meet timing guarantees depends on the speeds of the hardware's *processors*. The term "processor" is used in a general sense to denote any compute unit (*e.g.*, a CPU) that may execute at most one job at any time. A system containing one processor is a *uniprocessor*, while a system containing more than one processor is a *multiprocessor*. The scheduler must balance the demand of all jobs for the limited processor(s) such that sufficient processing capacity is provided to each job to meet its timing requirement.

There are three multiprocessor models generally studied by the real-time community:

- IDENTICAL, in which all processors have the same speed for all tasks (normalized to 1.0);
- UNIFORM, in which processor π_j has speed $sp^{(j)}$, which must be consistent for all tasks;
- and UNRELATED, in which each processor π_j may have a unique speed $sp^{(i,j)}$ for each task τ_i.

Under UNIFORM and UNRELATED, some related works make a simplifying assumption that the multiprocessor contains a limited number of distinct *types* of processors. Processors of the same type have an identical architecture, and hence, execution speeds. A multiprocessor with k distinct types is called a *k-type multiprocessor*.

Timing guarantees also depend on the choice of *scheduler*, code that decides when to execute jobs on processors. Most schedulers assign *priorities* to jobs and preferentially schedule higher-priority jobs. There exists a taxonomy of schedulers based on the behavior of priorities consisting of:

- *table-driven* (also, *static*) schedulers, which forgo priorities, instead deciding which processor should execute which job at any given time by looping over a hard-coded table of tasks and execution intervals (computed offline);
- *fixed-priority* schedulers, which assign all jobs of a given task the same priority;
- *job-level fixed-priority (JLFP)* schedulers, which assign each job its own *constant* priority;
- and *job-level dynamic-priority (JLDP)* schedulers, which allow the priority of any job to vary with time.

Some static schedulers use the term "frame" or "template" instead of table. Additionally, instead of a constant-length table, some schedulers shrink or stretch different iterations of the table depending on tasks' release times and deadlines. In this dissertation, all of these are also considered as table-driven schedulers.

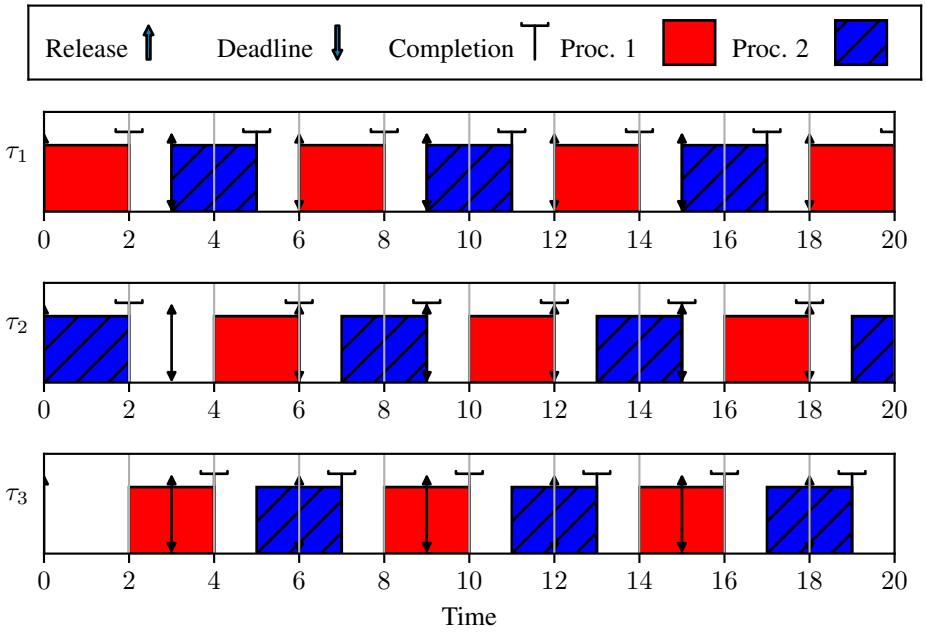
While fixed-priority schedulers seem to be more traditional in industry (possibly due to their simplicity and low overhead), certain JLFP and JLDP schedulers have been shown in prior work to have significantly higher efficacy with respect to SRT (Leontyev and Anderson, 2007). This dissertation exists in part to further this prior work.

Why care about SRT? We consider SRT due to certain advantages it has over HRT. First, SRT is suprisingly common in systems. Though HRT has been the more traditional sense of temporal correctness studied by the real-time community, an industry survey (Akesson et al., 2020) has shown that 67% of surveyed real-time systems contained some SRT component, while only 54% contained some HRT component. This highlights the prevalence of SRT in real industrial applications. Second, because SRT guarantees are less strict than HRT guarantees, SRT guarantees can often be met using less powerful hardware. Strict adherence to deadlines under HRT can require system designers to use hardware with processing capacity far greater than the long-run demand of the considered real-time applications. This is necessary to meet deadlines under worst-case conditions (*e.g.*, a large number of tasks with simultaneous job arrivals).

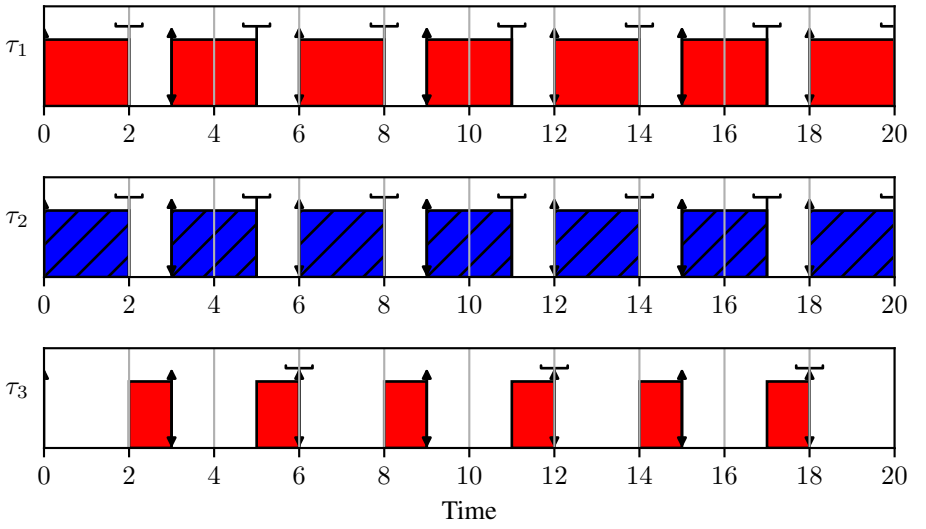
WC schedulers: great for SRT. This dissertation focuses on a class of JLDP schedulers called *window-constrained (WC)* schedulers that prior work (Leontyev and Anderson, 2007) has shown to be particularly effective for SRT. WC schedulers are derived from earliest-deadline-first (EDF),[1] a JLFP scheduler. Under EDF, jobs with earlier deadlines have higher priority.

▼ **Example 1.1.** This example demonstrates scheduling under EDF, as well as EDF's efficacy at lowering response times by comparing against a fixed-priority scheduler. Consider the task system of three tasks such that $(C_1, T_1) = (C_2, T_2) = (C_3, T_3) = (2.0, 3.0)$. Figure 1.1a depicts an EDF schedule of this system (assuming implicit deadlines) on an IDENTICAL multiprocessor with two processors. In this example, it is assumed that deadline ties are broken in favor of jobs belonging to lower-indexed tasks.

[1] When the considered platform is a multiprocessor, other works make distinctions such as between global EDF (G-EDF) and partitioned EDF (P-EDF) based upon which processors tasks are permitted to migrate to. These are both denoted EDF in this dissertation because the processors a task can execute on are specified by parameters of our multiprocessor model (see the upcoming Affinities paragraph).



(a) EDF schedule.



(b) Fixed-priority schedule.

Figure 1.1: SRT: EDF versus fixed-priority. (The legend depicted here is assumed also in later figures depicting schedules.)

Initially, at time 0, all three tasks have an arriving job. Because the system has implicit deadlines, the deadline of each of these jobs is the arrival time (0) offset by the corresponding tasks' periods (3.0). Because the deadlines of all executable jobs are equal, the jobs have equal priority, in which case the tie is broken in favor of jobs $\tau_{1,1}$ and $\tau_{2,1}$. Thus, these two jobs occupy the two processors until completion (at time 2.0). Once these two jobs complete, job $\tau_{3,1}$ becomes the highest-priority job (by virtue of being the only runnable job), and is executed on a processor starting at time 2.0.

At time 3.0, because the periods of all tasks is 3.0, all tasks have a second job arrival. Job $\tau_{1,2}$ occupies the available second processor because it executes no job. Job $\tau_{2,2}$, however, is unable to be scheduled because currently executing job $\tau_{3,1}$ has an earlier deadline (3.0) than $\tau_{2,2}$'s deadline (6.0).

When $\tau_{3,1}$ finishes at time 4.0, its second job $\tau_{3,2}$ has the same deadline as job $\tau_{2,2}$. The tie is resolved in favor of $\tau_{2,2}$, and it is scheduled on $\tau_{3,1}$'s former processor. The execution of job $\tau_{3,2}$ is delayed until job $\tau_{1,2}$ completes at time 5.0, thereby making its processor available.

Starting with the time interval $[3.0, 6.0)$, the schedule repeats every 3.0 time units (swapping π_1 and π_2 every 3.0 time units) following similar logic. Observe how the maximum response time of any job is 4.0 time units, which occurs for the jobs of task τ_3.

Compare these response times against those under the fixed-priority scheduler illustrated in Figure 1.1b. This example assumes jobs of tasks τ_1 and τ_2 have higher fixed priority than jobs of task τ_3. Whenever jobs of tasks τ_1 and τ_2 arrive every 3.0 time units, they occupy both processors because they have higher priority than any jobs of task τ_3. The response times of consecutive jobs of task τ_3 grow unboundedly (*e.g.*, job $\tau_{3,1}$ has response time 6.0, $\tau_{3,2}$ has response time 9.0, $\tau_{3,3}$ has response time 12.0, *etc.*). ▲

It was proven in a seminal work (Devi and Anderson, 2008) that under IDENTICAL, EDF satisfies a property called *SRT-optimality*, which means that if a system is *feasible* (*i.e.*, does not over-utilize its underlying hardware), then EDF guarantees bounded response times for all jobs. SRT-optimality was later extended to the class of WC schedulers under IDENTICAL (Leontyev and Anderson, 2007) and to UNIFORM for EDF (Yang and Anderson, 2017). WC schedulers are derived from EDF by allowing the time instant used to determine a job's priority (called the job's *priority point*) to vary within a bounded interval (hence, the "window" in window-constrained) around the implicit deadline.

SRT-optimality is a powerful property, as any system that over-utilizes the underyling hardware must have unbounded response times for some task under *any* scheduler. This makes EDF and its derived WC schedulers attractive for SRT applications.

1.1 Problem: EDF (and its Derivatives) are Poorly Understood

Fundamentally, **nobody** (especially the author) understands the SRT properties of EDF and its derived WC schedulers. There exist many critical open problems about EDF. This dissertation focuses on one such problem: to what extent does the SRT-optimality of EDF and WC schedulers extend to heterogeneous multiprocessors beyond IDENTICAL and UNIFORM? This question is especially relevant in a modern hardware context with multiprocessors growing increasingly heterogeneous. We briefly discuss sources of heterogeneity.

Heterogeneous multiprocessors. Heterogeneous multiprocessors contain more than one type of processor, and hence, do not follow the IDENTICAL model. Processors can differ by any number of factors including clock frequency, cache size, order of execution (*i.e.*, in instruction order or out of order), usage of pipelining/branch prediction, dynamic frequency and voltage scaling (DVFS),[2] data parallelism (SIMD and vector instructions), accelerators (*i.e.*, general purpose CPUs versus graphics processing units (GPUs), digital signal processors (DSPs), or field-programmable gate arrays (FPGAs)), *etc.*

Driving heterogeneity in multiprocessor design is a tradeoff between performance and efficiency. An exemplar of this heterogeneous multiprocessor design is the ARM big.LITTLE[3] architecture, which combines high-performance power-hungry "big" processors with slower energy efficient "LITTLE" processors. Higher performance by big processors is primarily achieved by higher clock frequency and larger per-processor cache size.

big.LITTLE multiprocessors are often treated analytically as UNIFORM multiprocessors, with relative processor speeds being determined experimentally. While the UNIFORM model is a fairly good first-order approximation, architectural differences should not cause uniform speedups among all tasks. For example, increasing per-processor cache size will have a disproportionate benefit for tasks with larger working set

[2]DVFS alone does not technically make a multiprocessor heterogeneous, as the processors may be identical architecturally. DVFS can cause differences at runtime as processors set different performance points.

[3]The big.LITTLE archicture was rebranded as and succeeded by the DynamIQ architecture by ARM in 2017. We continue to refer to this architecture as big.LITTLE to match with most related work and because certain evaluations in this work were performed on hardware that predated the change.

sizes, of which cache is a bottleneck on performance. Thus, UNRELATED, under which processor speeds depend on the executing task, is a more accurate multiprocessor model for heterogeneous architectures. This highlights the necessity of extending real-time analytical results to more complicated multiprocessor models such as UNRELATED.

Affinities. Besides differences in architecture, heterogeneity can arise via software with *affinities*. These are per-task restrictions on which processors a given task may execute on. In software, these are represented with affinity *masks*, bitmasks where each bit position corresponds with a processor. System designers may set affinities for reasons such as load balancing or increasing cache locality. Because, under affinities, the processors are not interchangeable from the perspective of the scheduler, the multiprocessor is heterogeneous.

We itemize seven cases for the affinities of a system, the former five of which have been studied extensively in the literature and the latter two being special cases relevant to this dissertation:

- GLOBAL, in which all tasks have affinity for all processors;
- PARTITIONED, in which each task has affinity for one processor;
- CLUSTERED, in which the processors are partitioned into *clusters* such that each task has affinity for the processors in one cluster;
- HIERARCHICAL, in which each affinity mask either has no intersection with, is a superset of, or is subset of each other affinity mask;
- ARBITRARY, in which any task may have any affinity;
- SEMI-PARTITIONED,[4] in which each task has affinity for either one or all processors;
- and SEMI-CLUSTERED, in which each task has affinity for either a cluster or all processors.

In this dissertation, we use "/" to denote that we are assuming a special case of affinities under the considered multiprocessor model. For example, ARBITRARY affinities under the IDENTICAL multiprocessor model is denoted as IDENTICAL/ARBITRARY. Assume that affinities are GLOBAL when affinities are not specified. Note that specifying affinities is redundant if the multiprocessor model under consideration is

[4]Note that some other works (*e.g.*, Kato and Yamasaki (2009); Afshar et al. (2012); Anderson et al. (2014); Voronov and Anderson (2018); Hobbs et al. (2021); Casini et al. (2021); *etc.*) dealing with mitigating migration overheads have alternative definitions of semi-partitioned.

UNRELATED. Under UNRELATED, a task not having affinity for a processor is analytically equivalent to said processor executing said task with speed 0.

Difficulty of heterogeneity. The difficulty of SRT analysis is proportional to the level of heterogeneity. Recall the informal definition of SRT-optimality detailed earlier: a scheduler is SRT-optimal if bounded response times are guaranteed so long as the system is feasible. The formal definition of feasible depends on the multiprocessor model considered, and grows in complexity with increasing heterogeneity. For example, for a uniprocessor, the feasibility condition is that the sum of all utilizations is at most 1.0. In contrast, for UNRELATED, the feasibility condition is that a particular linear program derived from the task system's parameters has a solution (Baruah, 2004). In SRT analysis, it is assumed that the considered system is feasible to prove response-time bounds. How to accomplish this is less clear with a more complicated feasibility condition.

`SCHED_DEADLINE`. `SCHED_DEADLINE` is an 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 (Gujarati et al., 2013, 2014; Scordino et al., 2018; Abeni et al., 2016; Lelli et al., 2016). This is also true of this dissertation, as work contained herein pertaining to affinities was motivated by a keynote talk given by Peter Zijlstra at ECRTS'17 (Zijlstra, 2017) on `SCHED_DEADLINE`. For these reasons, the analysis presented in this work was evaluated using `SCHED_DEADLINE` as a base.

An important aspect of `SCHED_DEADLINE` is its admission control system (ACS) The ACS has two purposes: (1) to limit the total processing capacity consumed by real-time tasks (by default, 5% of capacity must be reserved for non-real-time workloads at all times) and (2) to enforce SRT guarantees for real-time tasks. The ACS facilitates *dynamic task systems*, in which tasks may enter and leave the system over time. The ACS accomplishes its purposes by rejecting requests by tasks to enter the system if permitting their entrance would over-utilize the hardware.

The analysis by Devi and Anderson (2008) serves as the foundation for `SCHED_DEADLINE`'s migration logic and ACS. As such, SRT guarantees in the existing implementation are only maintained by the ACS under strictly IDENTICAL multiprocessors.

1.2 An Orthogonal Open Problem: Loose Response-Time Bounds

We take a brief aside to mention another open problem about EDF not addressed in this dissertation. We provide this aside to (1) provide a less biased view of the problems holding back EDF and its derivatives and (2) inform any future researcher working on EDF's SRT properties that the techniques presented in this dissertation are unlikely to make headway towards this problem.

Ever since the seminal work first demonstrating EDF's SRT-optimality (Devi and Anderson, 2008), it has been known that observable response times for synthetically generated tasks under EDF are much smaller than proven response-time bounds. It is the author's personal opinion that existing response-time bounds are too high for practical use, while observable response times (if bounds could be proven) are much more reasonable. At time of writing, deriving tight response-time bounds has been an open problem[5] for almost 20 years.

The techniques presented in this work seem orthogonal towards the problem of obtaining tight response-time bounds. This work is primarily interested in expanding SRT-optimality to a broader class of schedulers and multiprocessor models. This is done through abstractions that allow us to reason about simpler systems than a real-time task system and scheduler. Characteristics specific to EDF that result in low response times are seemingly not captured by these abstractions.

1.3 Thesis Statement

Lack of understanding of the SRT properties of EDF and its derived WC schedulers, particularly with respect to whether SRT-optimality properties extend to modern multiprocessors, is holding back the use of such schedulers for real-time applications. This dissertation aims to be a step towards this understanding, leading to the following thesis statement:

> *Heterogeneous multiprocessor models are required to describe many modern multiprocessors. The SRT-optimality of EDF and its derived schedulers can be extended to these models, such as IDENTICAL/ARBITRARY and UNRELATED, via scheduler variants targeted for such models. Though such variants may induce higher overheads than standard EDF on IDENTICAL, more practical implementations can be developed by restricting to special cases of these models.*

[5] Tight bounds can be computed under certain restrictions (Ahmed and Anderson, 2021), which will be discussed in Section 2.4.1

1.4 Contributions

The thesis is supported by the following contributions.

1.4.1 Improving SRT Analysis for UNIFORM and Extending to IDENTICAL/ARBITRARY

We derive polynomial response-time bounds for WC schedulers under UNIFORM (prior bounds (Yang and Anderson, 2017) were exponential and exclusive to EDF). We show that these same response-time bounds are also valid for WC schedulers under IDENTICAL/ARBITRARY, proving for the first time that WC schedulers are SRT-optimal under IDENTICAL/ARBITRARY. We also present unbounded response-time counterexamples for when jobs may be non-preemptive or when non-WC schedulers inspired by `SCHED_DEADLINE` are used.

1.4.2 WC Variant and Response-Time Bounds under UNRELATED

We derive response-time bounds under UNRELATED such that WC schedulers are asymptotically SRT-optimal (response-time bounds grow inversely as the task system approaches infeasibility).

1.4.3 Patching `SCHED_DEADLINE` for IDENTICAL/SEMI-PARTITIONED and 2-Type UNIFORM/SEMI-CLUSTERED

We demonstrate how the existing `SCHED_DEADLINE` implementation can have unbounded response times under heterogeneity. We present patches to restore provably[6] bounded response-times under IDENTICAL/SEMI-PARTITIONED multiprocessors and 2-type UNIFORM/SEMI-CLUSTERED multiprocessors such that each processor type is a cluster. We measure overhead increases due to these patches on hardware running synthetic workloads.

1.5 Organization

The rest of this dissertation is organized as follows. Background material is divided between Chapters 2 and 4. Chapter 2 covers theoretical background including the considered task and multiprocessor models, related work, and a brief review of related optimization problems. Chapter 3 covers all results involving response-time

[6]Assuming an idealized `SCHED_DEADLINE`. We do not attempt to formally verify the `SCHED_DEADLINE` code base.

bounds. Chapter 4 covers the SCHED_DEADLINE implementation. Chapter 5 covers our proposed patches to SCHED_DEADLINE for certain special cases of heterogeneous multiprocessors. Chapter 6 concludes.

CHAPTER 2: THEORETICAL BACKGROUND

This chapter covers considered models, related theoretical work, and relevant mathematical review.

2.1 Task Model

Time is assumed to be continuous. This dissertation considers the sporadic task model, described below. Notation is presented in the following definitions.

▽ **Definition 2.1.** The task system consists of n tasks denoted as $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$. △

It is assumed that each task is single-threaded, *i.e.*, never runs in parallel with itself.

▽ **Definition 2.2.** The task system runs on m processors $\pi = \{\pi_1, \pi_2, \ldots, \pi_m\}$. △

▽ **Definition 2.3.** Task τ_i has worst-case execution time C_i (relative to an execution speed of 1.0), period T_i, relative deadline D_i, and utilization $u_i \triangleq C_i/T_i$. For task τ_i, C_i, T_i, D_i, and u_i are all positive. △

We denote by a bracketed index $[i]$ that the corresponding parameter is the i^{th} largest of its kind. For example, the largest period and utilization are denoted $T_{[1]}$ and $u_{[1]}$, the smallest utilization is denoted $u_{[n]}$, and the fastest speed under UNIFORM is denoted $sp^{([1])}$.

▽ **Definition 2.4.** Task τ_i releases an infinite sequence of jobs with $\tau_{i,j}$ denoting the j^{th} job of task τ_i for $j \geq 1$. Job $\tau_{i,j}$ has *execution cost* $c_{i,j} \in (0, C_i]$, *arrival time* $a_{i,j}$, and *deadline* $d_{i,j} \triangleq a_{i,j} + D_i$. △

▽ **Definition 2.5.** Task τ_i is an *implicit-deadline* task if $D_i = T_i$. The *implicit deadline* of job $\tau_{i,j}$ is $\tilde{d}_{i,j} \triangleq a_{i,j} + T_i$. If τ_i is an implicit-deadline task, then $d_{i,j} = \tilde{d}_{i,j}$ for any job $\tau_{i,j}$. Note that $\tilde{d}_{i,j}$ is well-defined for job $\tau_{i,j}$ regardless of whether τ_i is an implicit-deadline task. △

▽ **Definition 2.6.** Task τ_i is a *constrained-deadline* task if $D_i \leq T_i$. △

Jobs are executed in order of arrival. Arrival times are separated such that for any $j \geq 1$, we have $a_{i,j} + T_i \leq a_{i,j+1}$.

▽ **Definition 2.7.** A task system is *periodic* if, for each job $\tau_{i,j}$, we have $a_{i,j} + T_i = a_{i,j+1}$. A task system is *sporadic* if, for each job $\tau_{i,j}$, we have $a_{i,j} + T_i \leq a_{i,j+1}$. △

▽ **Definition 2.8.** A task system is *synchronous* if, for each task τ_i, we have $a_{i,1} = 0$. △

▽ **Definition 2.9.** The *completion time* $f_{i,j}$ of job $\tau_{i,j}$ is the time instant that $\tau_{i,j}$ completes $c_{i,j}$ units of execution.

The *response time* of $\tau_{i,j}$ is the difference $f_{i,j} - a_{i,j}$. △

Additional notation is useful in our analysis for describing the job of a task that would, at a particular time instant, be executed if the task was scheduled.

▽ **Definition 2.10.** At time t, the *current* job of task τ_i is the incomplete job of task τ_i that has the earliest arrival time at time t.

We let $a_i(t)$, $d_i(t)$, $\tilde{d}_i(t)$ $c_i(t)$, and $rem_i(t)$ be the arrival time, deadline, implicit deadline, total execution cost, and remaining execution cost of the current job of τ_i at time t. Task τ_i's *deadline* and *implicit deadline* at time t are defined as $d_i(t)$ and $\tilde{d}_i(t)$, respectively. △

Recall from Definition 2.4 that we assume that each task releases an *infinite* sequence of jobs. Thus, for any task τ_i and time t, there are always jobs (that may not have arrived by time t) of task τ_i that are incomplete at time t. Because the current job of task τ_i at time t is the earliest of these jobs, the current job of task τ_i is well-defined at any time instant. Note that under Definition 2.10, the current job of a task τ_i at time t may not have arrived by time t.

Because the current job of task τ_i is well-defined at any time instant t, $a_i(t)$, $d_i(t)$, $c_i(t)$, and $rem_i(t)$ are also well-defined for any time instant t. This reduces the lengths of several proofs by omitting cases where the current job of a task at time t does not exist due to said task having already completed all of its jobs by time t. Note that our response-time analysis can still be applied to tasks that release finitely many jobs by assuming that the 'next' jobs of such tasks arrive arbitrarily far into the future.

Jobs must wait for their ready times to execute.

▽ **Definition 2.11.** A job $\tau_{i,j}$'s *ready time* $rdy_{i,j}$ is the time instant job $\tau_{i,j}$ (ignoring intra-task precedence constraints) first becomes eligible to execute. It is required that $rdy_{i,j} \leq a_{i,j}$, *i.e.*, the *ready time* of a job is at most its arrival time. Traditionally, for each job $\tau_{i,j}$, $rdy_{i,j} = a_{i,j}$, *i.e.*, a job becomes ready once it arrives. Allowing $rdy_{i,j} < a_{i,j}$ is called *early releasing*.

If, for job $\tau_{i,j}$, we have $t \geq rdy_{i,j}$, then job $\tau_{i,j}$ is *ready*. A task is *ready* if its current job is ready. The set of ready tasks at time t is denoted $\tau_{\text{rdy}}(t)$. △

Note that, if a task's current job is not ready, then the task cannot be scheduled, even if other jobs of the task are ready. This is because the current job of a task is the earliest (by arrival) incomplete job (Definition 2.10) and jobs of a task must complete in order of arrival.

Dynamic tasks. In systems such as `SCHED_DEADLINE`, tasks are dynamic in that they are expected to enter and leave the system over time. Tasks that are active (defined below) at a given time instant are those whose jobs contribute to load on the system at said time instant. Restrictions on the set of active tasks at any given time instant are necessary to prevent the system from being overloaded. The meaning of "overloaded" depends on the considered multiprocessor model, and will be formalized in Section 2.4. If new tasks becoming active would otherwise overload the system, said new tasks must wait for currently active tasks to become inactive, thereby allowing their consumed capacity to be used by new tasks.

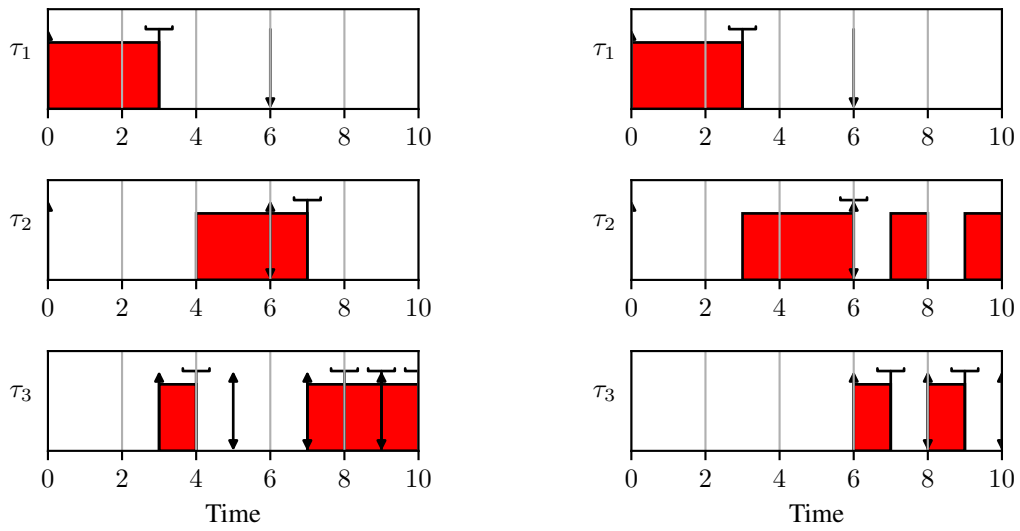
An active task becomes inactive by no longer releasing or executing jobs. In some instances, the time instant of the transition from active to inactive must be delayed past the completion time of a task's latest job. This is necessary for maintaining HRT guarantees for dynamic task systems, which will be demonstrated in Example 2.1. The exact time instant a task transitions from active to inactive is its last job's zero-lag time.[1]

▽ **Definition 2.12.** The *zero-lag time* of job $\tau_{i,j}$ is $zlt_{i,j} \triangleq a_{i,j} + \frac{c_{i,j}}{u_i}$. △

▽ **Definition 2.13.** At time t, a task τ_i is *active* if τ_i is ready or if task τ_i's most recently completed job is $\tau_{i,j}$ and $\tau_{i,j}$'s zero-lag time has not passed (*i.e.*, $t < zlt_{i,j}$). Task τ_i is *inactive* at t otherwise. The subset of τ of active tasks at t is $\tau_{\text{act}}(t)$. n_{act} denotes an upper bound on the maximum number of active tasks at any time instant. △

▼ **Example 2.1.** Consider the schedules illustrated in Figure 2.1. This example demonstrates that EDF meets deadlines for implicit-deadline tasks so long as total utilization is at most 1.0, but only if tasks wait until their latest jobs' zero-lag times before becoming inactive.

[1] Though our analysis, which considers SRT and not HRT, guarantees bounded response times even if tasks become inactive as soon as they complete their latest jobs, we delay the inactive transition time to the zero-lag time to conform with systems such as `SCHED_DEADLINE`. Be aware that our analysis at no point assumes that tasks remain active until their zero-lag times.



(a) Without waiting for $zlt_{1,1}$.

(b) Task τ_3 waits for $zlt_{1,1}$.

Figure 2.1: Deadline miss without waiting for zero-lag time.

Consider three implicit-deadline tasks τ_1, τ_2, and τ_3 such that $(C_1, T_1) = (C_2, T_2) = (3.0, 6.0)$ and $(C_3, T_3) = (1.0, 2.0)$. These tasks have $u_1 = u_2 = u_3 = 0.5$. The three tasks execute on a uniprocessor. Initially, at time 0, $\tau_{\text{act}}(0) = \{\tau_1, \tau_2\}$. The total utilization at time 0 is then $0.5 + 0.5 = 1.0$. The system is restricted such that the total utilization of active tasks at any time is at most 1.0.

In order for task τ_3 to become active, because total utilization is capped at 1.0, either task τ_1 or task τ_2 must first become inactive. Assume task τ_1 becomes inactive, upon which task τ_3 becomes active. Figure 2.1a illustrates the schedule where task τ_1 incorrectly becomes inactive as soon as it finishes its first job $\tau_{i,j}$, while Figure 2.1b illustrates the schedule where task τ_1 waits until $zlt_{1,1}$ before becoming inactive.

In Figure 2.1a, task τ_1 becomes inactive as soon as job $\tau_{1,1}$ completes at time 3.0. Task τ_3 becomes active at time 3.0, resulting in its first job $\tau_{3,1}$ arriving at time 3.0. Job $\tau_{3,1}$ has an earlier deadline (5.0) then that of job $\tau_{2,1}$ (6.0), so task τ_3 is scheduled over $[3.0, 4.0)$. Job $\tau_{2,1}$ completes at time 7.0, missing its deadline.

Compare this schedule to Figure 2.1b. The zero-lag time of job $\tau_{1,1}$ is $zlt_{1,1} = a_{1,1} + c_{1,1}/u_1 = 0 + 3.0/0.5 = 6.0$. Task τ_1 waits until time 6.0 before becoming inactive, thus task τ_3 does not becomes active until time 6.0. Job $\tau_{2,1}$, which missed its deadline in Figure 2.1a, meets its deadline in Figure 2.1b. ▲

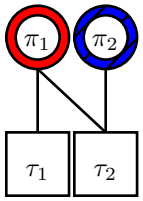


Figure 2.2: Affinity graph example.

The time instants when $\tau_{\text{act}}(t)$ changes are of interest because our analysis must show that the activation of new tasks is safe with respect to maintaining bounded response times.

▽ **Definition 2.14.** Let the *activation time instants* $(t_1^{\text{act}}, t_2^{\text{act}}, t_3^{\text{act}}, \dots)$ denote the increasing infinite sequence of time instants such that

- for $t \in (-\infty, t_1^{\text{act}})$, $\tau_{\text{act}}(t) = \emptyset$
- and for $t \in [t_k^{\text{act}}, t_{k+1}^{\text{act}})$, $\tau_{\text{act}}(t) = \tau_{\text{act}}\left(t_k^{\text{act}}\right)$ for each $k \in \mathbb{N}$. △

In words, the set of active tasks $\tau_{\text{act}}(t)$ changes only when $t = t_k^{\text{act}}$ for some $k \in \mathbb{N}$.

The worst-case total utilization of active tasks is a term in some of our response-time bounds.

▽ **Definition 2.15.** For subset $\tau' \subseteq \tau$, we let $U(\tau') \triangleq \sum_{\tau_i \in \tau'} u_i$. △

The worst-case total utilization is U_{max}, defined below.

▽ **Definition 2.16.** Let U_{max} denote an upper bound such that $\forall t :\ U_{\text{max}} \geq U(\tau_{\text{act}}(t))$. △

Affinities. Though affinity masks are represented as bitmasks in real systems such as Linux, it is convenient for analysis to represent them as sets or as graphs.

▽ **Definition 2.17.** Under a multiprocessor with affinities, the *affinity set* of task τ_i is denoted $\alpha_i \triangleq \{\pi_j \in \pi :\ \tau_i \text{ has affinity for } \pi_j\}$. △

▽ **Definition 2.18.** An *affinity graph* is a bipartite graph connecting a set of nodes corresponding to tasks to a set of nodes corresponding to processors. △

▼ **Example 2.2.** Consider a task system of τ_1 and τ_2 on two processors with affinities such that $\alpha_1 = \{\pi_1\}$ and $\alpha_2 = \{\pi_1, \pi_2\}$. The affinity graph for this system is illustrated in Figure 2.2. ▲

Affinity graphs are useful both for visualization and for allowing us to apply existing theorems and algorithms pertaining to bipartite graphs in analyzing systems with affinities.

Constant Bandwidth Server (CBS). Though our analysis assumes the sporadic task model, `SCHED_DEADLINE` instead employs CBS Abeni et al. (2015), whose parameters are analogous to those of sporadic tasks. A Server, as in CBS, is a real-time-systems term for a sequential container that prevents threads inside the container from over-consuming processor time, thereby starving other time-sensitive real-time tasks in the system. Most servers, including CBS, accomplish this via budgeting. Runnable threads inside the server consume the server's budget to execute until said budget is exhausted, at which point the server (and the threads within) becomes *throttled*.[2] The threads within the server must wait until the server's budget is replenished at a later time (usually once per *server period*) to execute again. Throttling limits the processor time the server can consume, thereby protecting other tasks in the system.

As with tasks, servers are assigned priorities and scheduled by the scheduler. A server is runnable when it has budget and contains a runnable thread. A server *suspends* once all of its contained threads have suspended (*i.e.*, waits or is blocked on a resource held by thread). A suspended server *wakes* once a contained thread becomes runnable again.

Pseudocode for CBS is presented in Algorithm 1, which we now detail. A CBS has parameters analogous to those of a sporadic task:

- `dl_runtime`, its maximum budget;
- `runtime`, its current budget;
- `dl_period`, its replenishment period;
- `dl_deadline`, its relative deadline;
- and `deadline`, its current server deadline.

For simplicity, we assume implicit deadlines (*i.e.*, `dl_deadline` = `dl_period`) when discussing CBS and in Algorithm 1. CBS will be covered in detail in Section 4.4.3 when discussing its implementation in `SCHED_DEADLINE`.

[2]Note that CBS as originally defined by Buttazzo and Abeni (1998) does not throttle on exhausting its budget, instead only reducing its priority. The description of CBS presented here follows the CBS implementation that does throttle.

```
1  begin
2      j ← 1
3      runtime′ ← dl_runtime
4      Replenish(now)
5      while True do
6          while runtime > 0 and not suspended do
7              Decrease runtime if executed
8          end while
9          if suspended then
10             wait for wakeup
11             if now ≥ deadline or
                runtime/(deadline − now) > dl_runtime/dl_deadline then
12                 Replenish(now)
13             else
14                 Arrival(now)
15             end if
16         end if
17         if runtime = 0 then
18             if now < deadline then
19                 wait for deadline
20             end if
21             Replenish(deadline)
22         end if
23     end while
24 end
25 Function Replenish(t):
26     Arrival(t)
27     runtime ← dl_runtime
28     deadline ← t + dl_period
29 end
30 Function Arrival(t):
31     if not suspended then
32         a_{i,j} ← t
33         c_{i,j−1} ← runtime′ − runtime
34         runtime′ ← runtime
35         j ← j + 1
36     end if
37 end
```

Algorithm 1: CBS pseudocode.

It would be natural to define CBS τ_i's replenishment times as $a_{i,j}$, `dl_runtime` as $c_{i,j}$, and the time instants where budget is exhausted (*i.e.*, `runtime` $= 0$) as completion times. This is insufficient due to suspensions, which are forbidden[3] under the sporadic task model considered in prior works on SRT-optimality (Devi and Anderson, 2008; Yang and Anderson, 2017)). Wakeups must thus also be treated as arrivals, and the corresponding preceding suspension times as completion times. The pseudocode in Algorithm 1 is thus supplemented with parameters

- j, the current job number;
- $a_{i,j}$ and $c_{i,j}$ as in Definition 2.4;
- and `runtime`$'$, the value of `runtime` at time $a_{i,j}$ that is used to compute $c_{i,j}$,

and function `Arrival`, which sets these values.

CBS τ_i is initialized in Lines 2-4, which set $a_{i,1}$ and provide CBS τ_i's initial budget and deadline (assume Line 33 does nothing when $j = 1$ because $c_{i,j-1}$ is undefined). It then begins budget tracking for the remainder of its lifetime.

Suspensions (Line 9) are more complicated. If the wakeup time following a suspension is past `deadline`, then CBS τ_i is replenished at the wakeup time (and not at `deadline`). This replenishment after the deadline is analogous to a sporadic task τ_i having job releases $a_{i,j+1} - a_{i,j} > T_i$.

If, on the other hand, the wakeup time occurs before `deadline`, there are different cases that govern how the CBS's parameters are modified. Note that these cases are designed to provide HRT guarantees on a uniprocessor or PARTITIONED multiprocessor, which is out of scope for this SRT-focused dissertation. As such, their design is not justified in this background chapter. Readers interested in the `SCHED_DEADLINE` CBS design are directed to the literature (Abeni et al., 2015).

On a wakeup prior to `deadline`, there are two cases depending on the *local density*

$$\frac{\texttt{runtime}}{\texttt{deadline} - \text{wakeup time}}.$$

In the case that local density $\leq$ `dl_runtime`/`dl_period`, the CBS continues executing with its current parameters (*e.g.*, `runtime` and `deadline`) are unchanged.

[3]Specifically, suspensions within jobs. Not being runnable due to waiting for the next job arrival is permitted.

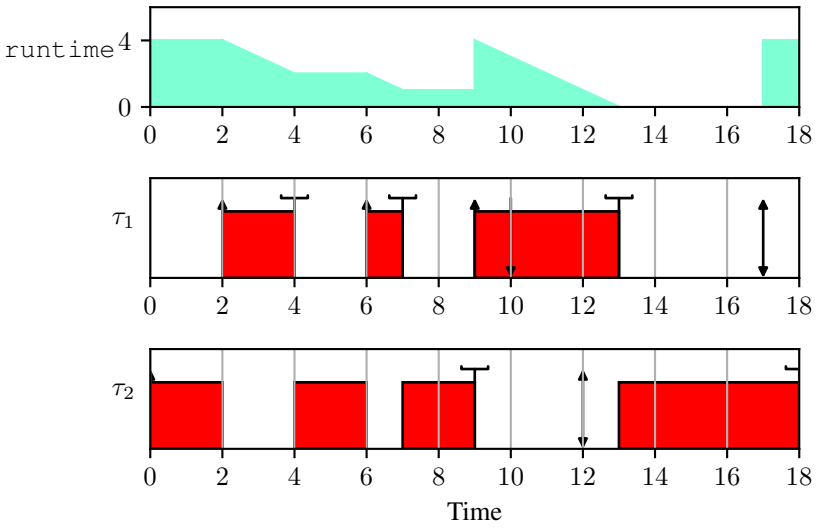


Figure 2.3: CBS example.

▼ **Example 2.3.** Consider CBS τ_1 with $(\texttt{dl\_runtime}, \texttt{dl\_period}) = (4.0, 8.0)$ and other workload τ_2 (which could be either another CBS or a sporadic task) executing on a uniprocessor. This system is illustrated in Figure 2.3. CBS τ_1 is initialized at time 2.0. This sets $a_{1,1} \leftarrow 2.0$ and $\texttt{deadline} \leftarrow 2.0 + \texttt{dl\_period} = 10.0$. CBS τ_1 executes for 2.0 time units over time interval $[2.0, 4.0)$, setting $\texttt{runtime} \leftarrow \texttt{runtime} - 2.0 = 2.0$.

At time 4.0, CBS τ_1 suspends until wakeup time 6.0. Because $\texttt{runtime}/\left(\texttt{deadline} - 6.0\right) = 2.0/\left(10.0 - 6.0\right) = 1.0/2.0 \leq 4.0/8.0 = \texttt{dl\_runtime}/\texttt{dl\_deadline}$, parameters $\texttt{runtime}$ and $\texttt{deadline}$ are unchanged (*i.e.*, the value of $\texttt{runtime}$ is continuous at time 4.0 and CBS τ_1 continues executing with $\texttt{deadline}$ of 10.0). This wakeup is treated as an arrival, setting $a_{1,2} \leftarrow 6.0$ and $c_{1,1} \leftarrow 2.0$. ▲

If the local density is greater than the CBS's density $\texttt{dl\_runtime}/\texttt{dl\_deadline}$, then the CBS is replenished at the wakeup time.[4] Note that because the wakeup occurred before $\texttt{deadline}$ in this case, this replenishment occurs less than $\texttt{dl\_period}$ time units from the previous replenishment. Because we count replenishments as job arrivals, the CBS does not follow the sporadic task model.

[4]This assumes deadlines are implicit. If deadlines are not implicit, there is a revised wakeup rule (Abeni et al., 2015) for the case that wakeup time $<$ $\texttt{deadline}$ and local density $>$ $\texttt{dl\_runtime}/\texttt{dl\_deadline}$. Under this revised rule, instead of replenishing on wakeup, budget $\texttt{runtime}$ is decreased such that the local density equals the CBS's density.

▼ **Example 2.3 (continued).** CBS τ_1 executes for 1.0 time unit over time interval $[6.0, 7.0)$, decreasing budget with $\texttt{runtime} \leftarrow \texttt{runtime} - 1.0 = 1.0$, before suspending again at time 7.0 until wakeup at time 9.0. At the wakeup at time 9.0, $\texttt{runtime}/\left(\texttt{deadline} - 9.0\right) = 1.0/\left(10.0 - 9.0\right) = 1.0 > 4.0/8.0 = \texttt{dl\_runtime}/\texttt{dl\_deadline}$. This triggers an early (*i.e.*, before $\texttt{deadline} = 10.0$) replenishment at time 9.0, setting $\texttt{runtime} \leftarrow \texttt{dl\_runtime} = 4.0$ and $\texttt{deadline} \leftarrow 9.0 + \texttt{dl\_period} = 17.0$. This replenishment sets $a_{1,3} \leftarrow 9.0$ and $c_{1,2} \leftarrow 1.0$. ▲

If budget is exhausted (Line 17), it waits for the next replenishment time `deadline` if `deadline` is in the future (*i.e.*, the CBS is throttled). Otherwise, if `deadline` has passed (recall it is possible to execute past a deadline in a SRT system), the replenishment is applied retroactively at `deadline`.

▼ **Example 2.3 (continued).** CBS τ_1 then executes over $[9.0, 13.0)$, setting $\texttt{runtime} \leftarrow \texttt{runtime} - 4.0 = 0$. CBS τ_1 is throttled due to running out of budget at time 13.0 and waits for replenishment at $\texttt{deadline} = 17.0$. At this replenishment at time 17.0, $\texttt{runtime} \leftarrow \texttt{dl\_runtime} = 4.0$, $\texttt{deadline} \leftarrow 17.0 + \texttt{dl\_period} = 25.0$, $a_{1,4} \leftarrow 17.0$, and $c_{1,3} \leftarrow 4.0$. ▲

2.2 Scheduler Classifications

This section presents abstractions of schedulers. The following assumption is made about schedulers discussed in this dissertation.

▷ **Non-Fluid Assumption.** Schedulers are assumed to be *non-fluid*, *i.e.*, at any time t, there exists $t_+ > t$ and $t_- < t$ such that for any job $\tau_{i,j}$

- if job $\tau_{i,j}$ is scheduled on processor π_j at time t, then job $\tau_{i,j}$ is scheduled on processor π_j over the interval $[t, t_+)$;
- if job $\tau_{i,j}$ is unscheduled at time t, then job $\tau_{i,j}$ is unscheduled over $[t, t_+)$;
- if job $\tau_{i,j}$ is scheduled on processor π_j at time t_-, then job $\tau_{i,j}$ is scheduled on processor π_j over the interval $[t_-, t)$;
- and if job $\tau_{i,j}$ is unscheduled at time t_-, then job $\tau_{i,j}$ is unscheduled over $[t_-, t)$. ◁

Informally, the Non-Fluid Assumption states that the multiprocessor does not reschedule with infinite frequency. This is a given for any practical scheduler.

Task priorities are abstracted as priority points.

▽ **Definition 2.19.** (Def. 1 of Leontyev and Anderson (2007)) Associated with each job $\tau_{i,j}$ is a function $pp_{i,j}(t)$, called its *priority point function*. If, at time t, $pp_{i,j}(t) < pp_{h,k}(t)$ holds, then the priority of job $\tau_{i,j}$ is higher than that of job $\tau_{h,k}$ at t.

The *priority point* of task τ_i with current job $\tau_{i,j}$ is $pp_i(t) \triangleq pp_{i,j}(t)$. △

For example, EDF can be defined as the scheduler such that for any job $\tau_{i,j}$, $pp_{i,j}(t) = d_{i,j}$.

WC schedulers are schedulers such that the priority point $pp_i(t)$ always lies within a bounded window around task τ_i's implicit deadline.

▽ **Definition 2.20.** A scheduler is *window constrained (WC)* with *priority window* $\phi \geq 0$ if, for any time t and task τ_i, we have $\left|pp_i(t) - \tilde{d}_i(t)\right| \leq \phi$. △

For example, under EDF, we have

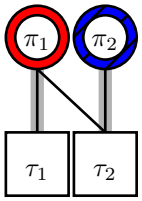
$$\begin{aligned}
\left|pp_i(t) - \tilde{d}_i(t)\right| &= \{\text{Under EDF, } pp_i(t) = d_i(t)\} \\
&\quad \left|d_i(t) - \tilde{d}_i(t)\right| \\
&= \left|a_i(t) + D_i - \tilde{d}_i(t)\right| \\
&= \{\text{Definition 2.5}\} \\
&\quad |a_i(t) + D_i - a_i(t) - T_i| \\
&= |D_i - T_i|.
\end{aligned}$$

Thus, EDF with arbitrary (but finite) relative deadlines is WC with $\phi = |D_i - T_i|$.

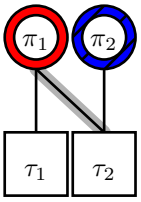
While at first glance, a scheduler is fully defined by how $pp_i(t)$ is defined for each task τ_i, such a definition can be ambiguous on sufficiently heterogeneous systems.

Let the term *configuration* denote some assignment of tasks to processors that may be selected by a scheduler at some time instant.

▼ **Example 2.4.** Consider the two configurations (highlighted edges indicate assignment of a task to a processor) of task system $\tau = \{\tau_1, \tau_2\}$ executing on a two processor IDENTICAL/ARBITRARY system illustrated in Figure 2.4. Suppose at some time t, both tasks τ_1 and τ_2 are ready and $pp_2(t) < pp_1(t)$, *i.e.*, τ_2's job is of higher priority than τ_1's job.



(a) Configuration where tasks τ_1 and τ_2 are both scheduled.



(b) Configuration where only task τ_2 is scheduled.

Figure 2.4: Two configurations.

Priority points are followed by both the configurations in Figure 2.4a and Figure 2.4b because no unscheduled task can preempt a lower-priority scheduled task under either configuration. Under these priority points, which of the illustrated configurations should be chosen by the scheduler at time t? While that of Figure 2.4a is superior because both tasks are scheduled instead of only task τ_2, a scheduler implementation may prefer that of Figure 2.4b if, for example, τ_2 is cache-hot on processor π_1.

The question of which configuration to choose can be further complicated by additional heterogeneity. Suppose that instead of being IDENTICAL processors, the system is UNIFORM such that processor π_2 is a slow processor. Is the configuration in Figure 2.4a still superior merely because both tasks are scheduled, especially if task τ_2 has utilization $u_2 > sp^{(2)}$? ▲

The ambiguity illustrated in Example 2.4 over which configuration to select results in scheduler *variants*. A scheduler variant is defined by which configurations it may choose given the priorities of jobs. For example, when discussing related work later in this chapter, we will cover Strong-APA-EDF and Weak-APA-EDF, EDF variants specified for IDENTICAL/ARBITRARY. Strong-APA-EDF requires that the number of scheduled tasks is always maximized (*e.g.*, in Example 2.4, choosing the configuration in Figure 2.4a), while Weak-APA-EDF permits any configuration such that a lower-priority task is never scheduled on a processor that an unscheduled higher-priority task has affinity for (*e.g.*, either of the configurations may be chosen). The choice of variant is critically important for SRT-optimality. We will show in Chapter 3 that Strong-APA-EDF is SRT-optimal, while Weak-APA-EDF is not.

It will also be useful to have notation for the speed a task is currently executing at under a considered scheduler.

▽ **Definition 2.21.** For task τ_i and time t, the *current speed* $csp_i(t)$ is the speed of execution of task τ_i at time t under the considered scheduler. △

The current speed of a task τ_i is 0 if task τ_i is not assigned a processor in the configuration chosen by the scheduler at time t. If τ_i is assigned a processor π_j, then $csp_i(t) = 1.0$ under IDENTICAL, $csp_i(t) = sp^{(j)}$ under UNIFORM, and $csp_i(t) = sp^{i,j}$ under UNRELATED.

2.3 Optimization Review

In this dissertation and in prior work to be discussed in Section 2.4, scheduler variants are defined as choosing configurations based on the solution to some optimization problem defined by task priorities and processor speeds. This subsection covers optimization problems corresponding with these scheduler variants. Framing scheduler variants as optimization problems simplifies analysis by allowing us to take advantage of theorems proven about said optimization problems. Whether these optimization problems can be solved efficiently also impacts how efficiently various scheduler variants can be implemented.

Optimization problems AP and MVM, discussed below, take as input bipartite graphs. Because, when we reference these optimization problems, the nodes of these bipartite graphs will correspond with tasks and processors, we also denote these node sets as τ and π with respective sizes n and m.

Assignment Problem (AP). An AP$(\tau, \pi, \mathbf{P})$ instance with *profit matrix* $\mathbf{P} \in \mathbb{R}^{n \times m}$ solves for

$$\max \sum_{\tau_i \in \tau} \sum_{\pi_j \in \pi} p_{i,j} \cdot x_{i,j} \text{ such that} \tag{2.1}$$

$$\forall \tau_i \in \tau : \sum_{\pi_j \in \pi} x_{i,j} \leq 1.0 \tag{2.2}$$

$$\forall \pi_j \in \pi : \sum_{\tau_i \in \tau} x_{i,j} \leq 1.0 \tag{2.3}$$

$$\forall \tau_i \in \tau : \forall \pi_j \in \pi : x_{i,j} \in \{0, 1\}. \tag{2.4}$$

In general, AP instances are used to match workers (τ) to jobs—jobs in the general sense, not real-time jobs—(π) such that each worker works on at most one job (2.2) and each job is worked on by at most one worker (2.3). $x_{i,j} = 1$ holds if the i^{th} worker works on the j^{th} job and $x_{i,j} = 0$ holds otherwise. Element $p_{i,j}$ of $\mathbf{P}$ represents the profit generated from the i^{th} worker completing the j^{th} job. Constraint (2.4) forces the decision variables $x_{i,j}$ to be binary.

Despite being expressed as an integer linear program (ILP), an AP instance can be solved in polynomial time using an algorithm such as the Hungarian method (Edmonds and Karp, 1972), though such algorithms

exceed quadratic time complexity. An AP instance can be solved more efficiently by leveraging the solution of a smaller instance. Formally, if a solution for instance $\mathrm{AP}(\tau, \pi, \mathbf{P})$ is known, then the *incremental* AP instance

$$\mathrm{AP}\left(\tau \cup \{\tau_{n+1}\}, \pi \cup \{\pi_{m+1}\}, \begin{bmatrix} & & \begin{bmatrix} p_{1,m+1} \\ p_{2,m+1} \\ \vdots \\ p_{n,m+1} \end{bmatrix} \\ & \mathbf{P} & \\ & & \\ \begin{bmatrix} p_{n+1,1} & p_{n+1,2} & \cdots & p_{n+1,m} \end{bmatrix} & & p_{n+1,m+1} \end{bmatrix}\right)$$

can be solved in $O\left(\max\{n, m\}^2\right)$ time (Toroslu and Üçoluk, 2007). The algorithm presented by Toroslu and Üçoluk for an incremental AP instance is conceptually similar to a single iteration of the Hungarian method.

The following theorem about AP instances will be used in the analysis in Chapter 3.

▷ **Theorem 2.1 (Theorem 3.2.1 by Matoušek and Gärtner (2007)).** An optimal solution of an AP instance remains optimal even if (2.4) is relaxed to $x_{i,j} \geq 0$, *i.e.*, the integer constraint is relaxed to a non-negativity constraint. ◁

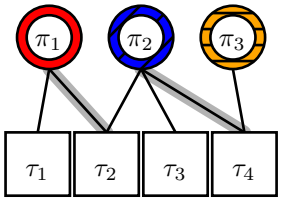
Maximum Vertex-Weighted Matching (MVM). Discussing MVM instances requires the following definitions from graph theory.

▽ **Definition 2.22.** A *matching* $\mathbb{M}$ in a graph is a subset of edges in the graph such that no edges share a common vertex. △

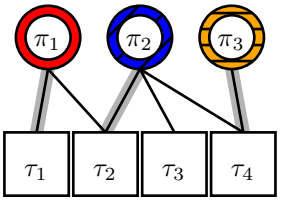
▽ **Definition 2.23.** A matching $\mathbb{M}$ is *maximal* if any other matching on the graph matches equal or fewer vertices. △

▽ **Definition 2.24.** An *alternating path* is a path in a graph with matching $\mathbb{M}$ such that edges in the path alternate between being not in and in $\mathbb{M}$. △

▽ **Definition 2.25.** An *augmenting path* is an alternating path that begins and ends with unmatched vertices. △



(a) Matching with augmenting path.



(b) Maximal matching.

Figure 2.5: Matching example.

▼ **Example 2.5.** Consider the affinity graph illustrated in Figure 2.5. Figure 2.5a illustrates matching $\mathbb{M}_1 = \{(\tau_2, \pi_1), (\tau_4, \pi_2)\}$. Path $(\tau_1, \pi_1, \tau_2, \pi_2, \tau_4, \pi_3)$ is an alternating path because $(\tau_1, \pi_1) \notin \mathbb{M}_1$, $(\tau_2, \pi_1) \in \mathbb{M}_1$, $(\tau_3, \pi_2) \notin \mathbb{M}_1$, $(\tau_4, \pi_2) \in \mathbb{M}_1$, $(\tau_4, \pi_3) \notin \mathbb{M}_1$. This alternating path is an augmenting path because it begins with τ_1 such that $(\tau_1, \pi_1) \notin \mathbb{M}_1$ and ends with π_3 such that $(\tau_4, \pi_3) \notin \mathbb{M}_1$.

The matching $\mathbb{M}_2 = \{(\tau_1, \pi_1), (\tau_2, \pi_2), (\tau_4, \pi_3)\}$ that arises from inverting edges in the augmenting path is illustrated in Figure 2.5b.

Matching $\mathbb{M}_2$ is maximal because no other matching can pair more vertices than $\mathbb{M}_2$. This can be seen in Figure 2.5b in that all three processors are already matched. Matching $\mathbb{M}_1$ is not maximal because $\mathbb{M}_2$ matches more vertices. ▲

▽ **Remark 1.** Consider two matchings $\mathbb{M}_1$ and $\mathbb{M}_2$ such that $\mathbb{M}_2$ arises from inverting an augmenting path in $\mathbb{M}_1$. Any task τ_i or processor π_j matched in matching $\mathbb{M}_1$ is also matched in $\mathbb{M}_2$. △

Remark 1 can be observed in Figure 2.5.

An MVM$\left(\tau, \pi, \overrightarrow{\psi}, \mathbb{E}\right)$ instance with *profit vector* $\overrightarrow{\psi} \in \mathbb{R}^n$ and edge set $\mathbb{E} \subseteq \tau \times \pi$ solves for

$$\max \sum_{\tau_i \in \tau} \psi_i \sum_{(\tau_i, \pi_j) \in \mathbb{E}} x_{i,j} \text{ such that} \tag{2.5}$$

$$\forall \tau_i \in \tau : \sum_{\pi_j \in \pi} x_{i,j} \leq 1 \tag{2.6}$$

$$\forall \pi_j \in \pi : \sum_{\tau_i \in \tau} x_{i,j} \leq 1 \tag{2.7}$$

$$\forall \tau_i \in \tau : \ \pi_j \in \pi : x_{i,j} \in \{0, 1\}.$$

MVM is a special case of AP where $p_{i,j} = \psi_i$ if $(\tau_i, \pi_j) \in \mathbb{E}$ and $p_{i,j} = 0$ otherwise. In the language of workers and jobs, the profit yielded for completing any job depends only on the corresponding worker, but each worker can only complete certain jobs.

Note that the binary matrix $\mathbf{X}$ that solves an AP or MVM instance is an alternative representation of a matching $\mathbb{M}$ such that $x_{i,j} = 1$ when edge $(\tau_i, \pi_j) \in \mathbb{M}$ and $x_{i,j} = 0$ when edge $(\tau_i, \pi_j) \notin \mathbb{M}$. Constraints (2.2) and (2.6) prevent any task from being matched to more than one processor and constraints (2.3) and (2.7) prevent any processor from being matched to more than one task.

▼ **Example 2.6.** Consider the matchings illustrated in Figure 2.5. Matrices

$$\mathbf{X}_1 = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \text{ and } \mathbf{X}_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

are alternative representations of matching $\mathbb{M}_1$ in Figure 2.5a and $\mathbb{M}_2$ in Figure 2.5b, respectively. ▲

▽ **Remark 2.** A matching that optimally solves an MVM instance contains no augmenting paths beginning with a task τ_i with $\psi_i > 0$. △

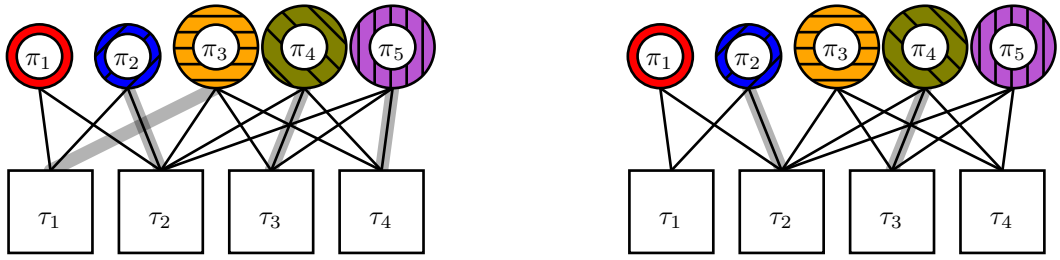
The remark follows from Remark 1. Otherwise, if an augmenting path beginning with some task τ_i exists, the objective function value of the MVM instance (see (2.5)) can be increased by ψ_i using the matching that arises from inverting along the augmenting path.

The ensuing theorem relates augmenting paths with maximal matchings.

▷ **Theorem 2.2 (Theorem 1 by Berge (1957)).** A matching $\mathbb{M}$ of a graph is maximal if and only if there is no augmenting path for $\mathbb{M}$ in the graph. ◁

Remark 2 and Theorem 2.2 imply that the optimal solution of any MVM instance is a maximal matching. This fact will be used in the analysis in Chapter 3 on systems with affinities.

Configurations. Any configuration can be represented as a binary matrix $\mathbf{X}$ satisfying (2.2) and (2.3) (or equivalently, satisfying (2.6) and (2.7)) or its corresponding matching. For example, $\mathbf{X}_1$ in Example 2.6 represents a configuration where task τ_2 is scheduled on processor π_1 and task τ_4 is scheduled on processor



(a) Configuration $\mathbf{X}^{(1)}$.

(b) Configuration $\mathbf{X}^{(2)}$.

Figure 2.6: Non-canonical and canonical configurations. (In this and in later affinity graphs, faster processors have larger sizes.)

π_2 (recall Figure 2.5a). Not every $\mathbf{X}$ satisfying (2.2) and (2.3) represents a valid configuration at every time instant. For example, at some time t, we may have $x_{i,j} = 1$ while $\tau_i \notin \tau_{\text{rdy}}(t)$, *i.e.*, task τ_i is not ready at time t. We call matrix $\mathbf{X}$ canonical at time t if it corresponds to a valid configuration.

▽ **Definition 2.26.** Matrix $\mathbf{X} \in \mathbb{R}^{n \times m}$ is *canonical* at time t if, for any task $\tau_i \in \tau$,

- τ_i is matched in $\mathbf{X}$ to a processor only if τ_i is ready at t, *i.e.*, $\tau_i \notin \tau_{\text{rdy}}(t) \Rightarrow \forall \pi_j \in \pi : \; x_{i,j} = 0$.
- and τ_i is not matched to a processor it does not have affinity for, *i.e.*, $\pi_j \notin \alpha_i \Rightarrow x_{i,j} = 0$. △

▼ **Example 2.7.** Consider a system of four tasks and five CPUs with affinities as illustrated in Figure 2.6. Suppose that at some time t, we have $\tau_{\text{rdy}}(t) = \{\tau_1, \tau_2, \tau_3\}$. Consider the configurations

$$\mathbf{X}^{(1)} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \text{ and } \mathbf{X}^{(2)} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Configuration $\mathbf{X}^{(1)}$ (Figure 2.6a) is not canonical at time t because task $\tau_4 \notin \tau_{\text{rdy}}(t)$ is matched with processor π_5 (*i.e.*, $x^{(1)}_{4,5} = 1$) and because task τ_1 is matched to processor $\pi_3 \notin \alpha_1$ (*i.e.*, $x^{(1)}_{1,3} = 1$ and τ_1 and π_3 are not connected by an edge in the affinity graph in Figure 2.6a). Configuration $\mathbf{X}^{(2)}$ (Figure 2.6b) is a duplicate of $\mathbf{X}^{(1)}$ that is made canonical at t by setting $x^{(2)}_{4,5} = 0$ and $x^{(2)}_{1,3} = 0$. ▲

Recall from the discussion at the beginning of Section 2.3 that some scheduler variants that will be discussed in this dissertation are defined as choosing a configuration that corresponds to some optimal

solution of an MVM or AP instance at every time instant. Keep in mind that, for such scheduler variants to be well-defined, we must demonstrate that there always exists an optimal solution that is canonical, as otherwise our scheduler variant may be required to either schedule non-ready tasks or violate tasks' affinities.

Going forward, whenever a matrix $\mathbf{X}$ is referred to as a configuration at some time t, interpret this to mean that $\mathbf{X}$ is canonical at t.

2.4 Related Work

Much of the related work referenced by our analysis pertains to feasibility.

▽ **Definition 2.27.** A task system is *feasible* on a considered multiprocessor if there exists *some* schedule such that each job completes by its deadline. △

Note that the feasibility conditions presented in the following subsections assume that all tasks have implicit deadlines. Our analysis in Chapter 3 will share this assumption. Because EDF with non-implicit-deadline tasks is a WC scheduler, our analysis (which considers all WC schedulers) can be used to compute response-time bounds even if tasks do not have implicit deadlines.

2.4.1 Work under IDENTICAL

A majority of SRT analysis of EDF considers IDENTICAL multiprocessors. The feasibility condition under IDENTICAL is as follows.

▷ **IDENTICAL-Feasibility (Srinivasan and Anderson, 2006).** Task system τ is feasible if and only if

$$U(\tau) \leq m, \text{ and } \forall \tau_i \in \tau : \ u_i \leq 1.0. \qquad \triangleleft$$

Devi and Anderson (2008) originally proved that for any sporadic IDENTICAL-Feasible system under EDF, task τ_i has response time at most

$$T_i + \frac{\sum_{k=1}^{m-1} C_{[k]} - C_{[n]}}{m - \sum_{k=1}^{m-1} u_{[k]}} + C_i. \tag{2.8}$$

Expression (2.8) is $O\left(m \cdot C_{[1]}\right)$ when tasks are heavy (*i.e.*, many tasks τ_i have $u_i \approx 1.0$), despite *observed* response times for synthetically generated task systems being $O\left(T_{[1]} + C_{[1]}\right)$ (Devi and Anderson, 2008).

The response-time bound (2.8) is proven inductively: for a job of interest $\tau_{i,j}$, assuming every job with earlier (and equal with favored tie breaking) deadline than $d_{i,j}$ has response time at most that in (2.8), then the response time of $\tau_{i,j}$ is also at most that in (2.8).

Additionally, Devi and Anderson proved a slightly higher response-time bound for sporadic IDENTICAL-Feasible systems where jobs of tasks are non-preemptive (*i.e.*, jobs cannot be preempted or migrated once they begin executing on a processor). Non-preemptivity is valuable for purposes including critical sections, synchronization, and reducing execution costs (due to reduced context-switch overhead). As such, the proof by Devi and Anderson raised the question of whether EDF retains SRT-optimality with non-preemptive jobs on heterogeneous multiprocessors.

Leontyev and Anderson (2007) extended the result by Devi and Anderson by showing that the class of WC schedulers has bounded response times for any sporadic IDENTICAL-Feasible system.[5] Leontyev and Anderson (2007) further observed that, under IDENTICAL and without early releasing, any non-preemptive WC scheduler can itself be abstracted as a fully-preemptive WC scheduler (with increased ϕ_i). Setting $pp_{i,j}(t) \leftarrow a_{i,j} - \max_{\tau_k \in \tau} \{\phi_k\}$ once job $\tau_{i,j}$ is scheduled implies that any newly released job has lower priority than job $\tau_{i,j}$, thus, job $\tau_{i,j}$ cannot be preempted.

Orthogonal to the work by Leontyev and Anderson, which expanded the class of SRT-optimal schedulers, has been work to reduce the response-time bound (2.8) by Devi and Anderson. Erickson et al. (2010) developed an analytical technique they called *compliant-vector analysis (CVA)*. At a high level, instead of assuming the bound in (2.8), CVA performs induction assuming a similar bound where the fractional term in (2.8) is allowed to be task-dependent. CVA was later supplemented with the WC Fair-Lateness scheduler (Erickson and Anderson, 2012) that uses linear programming to identify optimal relative priority points (*i.e.*, $pp_{i,j}(t) = a_{i,j} + y_i$ for some task-dependent offset y_i) for reducing maximum or average maximum response times bounds.[6] CVA with Fair-Lateness reduces analytical response-time bounds from that in (2.8), but bounds remain in the same order of magnitude (*i.e.*, $O\left(m \cdot C_{[1]}\right)$).

The state-of-the-art response-time bounds for EDF under IDENTICAL were proven by Valente (2016). These bounds are fairly complex, containing several instances of the $\max$ function whose arguments are the

[5]Note that the definition of a WC scheduler used by Leontyev and Anderson is differently parameterized than Definition 2.20. Unlike Definition 2.20, which uses the single parameter ϕ, Leontyev and Anderson define unique per-task $\psi_i \in \mathbb{R}_{\geq 0}$ and $\phi_i \in \mathbb{R}_{\geq 0}$ such that $pp_{i,j}(t) \in [a_{i,j} - \phi_i, \tilde{d}_{i,j} + \psi_i]$. Any scheduler that is WC under one definition is also WC under the other.

[6]Note that Fair-Lateness schedulers actually optimize for *lateness* bounds, $\max_{\tau_i \in \tau} \max_{j \in \mathbb{N}} \{f_{i,j} - d_{i,j}\}$, instead of response-time bounds. The two bounds are roughly proportional.

subsets of τ, which number exponential in n. There is no known polynomial-time algorithm for computing these bounds. These bounds are roughly logarithmic with m. A more detailed description of the asymptotic behavior is difficult to derive due to the complexity of the bounds.

Many works have proven lower response-time bounds by making additional assumptions about the task model. Ahmed and Anderson (2021) considered synchronous periodic *pseudo-harmonic* task systems. Pseudo-harmonic systems are such that the least common multiple of all periods is equivalent to the largest period $T_{[1]}$. Under this restricted task model, Ahmed and Anderson were able to prove exact $O(T_{[1]})$ response-time bounds that take pseudo-polynomial time to compute. These bounds concern EDF-like schedulers, a subclass of WC that requires $pp_{i,j}(t)$ to be constant for each job $\tau_{i,j}$. Recently, Buzzega et al. (2023) further restricted the task model to synchronous periodic *uniform-instance* tasks (not to be confused with UNIFORM multiprocessors). Uniform-instance tasks are such that each task τ_i has equivalent C_i and T_i, *i.e.*, $C_i = C$ and $T_i = T$ for some C and T. For any such task system, it was proven that any response time is at most $T + C$.

An alternative method of reducing response-time bounds has been to introduce intra-task parallelism to the task model. Under the *no-precedence-constraints (NPC)-sporadic task model* (named by Yang and Anderson (2014)), the ready time of any job $\tau_{i,j}$ is allowed to precede the completion time of job $\tau_{i,j-1}$ (*i.e.*, unlike Definition 2.11, it is only required that $rdy_{i,j} \leq a_{i,j}$ under early releasing and $rdy_{i,j} = a_{i,j}$ if not). Erickson and Anderson (2011) proved that for implicit-deadline NPC-sporadic tasks, for any IDENTICAL-Feasible system, the response time of any job of any task τ_i is $O\left(T_i + C_{[1]} + C_i\right)$. Note that IDENTICAL-Feasible for NPC-sporadic tasks only requires that $\sum_{\tau_i \in \tau_i} u_i \leq m$. It is no longer required that for any task τ_i we have $u_i \leq 1.0$. Amert et al. (2019) defined the *restricted-parallelism (RP)-sporadic task model* such that NPC-sporadic and standard sporadic task models are special cases. RP-sporadic tasks are parameterized by a per-task parallelism level that varies from one (standard sporadic task) to m (NPC-sporadic task). The re-addition of restrictions on parallelism, however, results in response-time bounds again being $O\left(m \cdot C_{[1]}\right)$.

2.4.2 Work under IDENTICAL/ARBITRARY

There has not been much focus on SRT analysis under ARBITRARY affinities. A well-studied special case is semi-PARTITIONED schedulers, which are slightly distinct from the notion of SEMI-PARTITIONED (see Chapter 1) in this dissertation. Semi-PARTITIONED schedulers set affinities, usually to reduce migrations. Such schedulers usually begin with an offline partitioning step that assigns most tasks affinity for a single

processor and the few (usually $O(m)$) remaining tasks affinity for multiple (usually two) processors. Semi-PARTITIONED schedulers embodying this approach for SRT include EDF-os (Anderson et al., 2014) and EDF-sc[7] (Hobbs et al., 2021), which are both SRT-optimal (with the corresponding feasibility condition being IDENTICAL-Feasible). Though named after and derived from EDF, we do not consider such schedulers as EDF variants because they usually employ some level of hierarchical scheduling (*e.g.*, migrating tasks have statically higher priority than fixed tasks, regardless of deadline). Such semi-PARTITIONED works also differ from this dissertation because we assume affinities are specified outside of the scheduler. We assume that at most, the scheduler can require that affinities have certain structure (such as SEMI-PARTITIONED or SEMI-CLUSTERED), but has no say on the affinities of specific tasks.

The feasibility condition under IDENTICAL/ARBITRARY is as follows.

▷ **IDENTICAL/ARBITRARY-Feasibility (Baruah and Brandenburg, 2013).** Task system τ is feasible if and only if $\forall \tau_i \in \tau : \ u_i \leq 1.0$ and

$$\forall \tau_i \in \tau : \ \sum_{\pi_j \in \alpha_i} x_{i,j} = 1.0, \text{ and} \tag{2.9}$$

$$\forall \pi_j \in \pi : \ \sum_{\tau_i \in \tau} x_{i,j} \cdot u_i \leq 1.0. \tag{2.10}$$

for some $\mathbf{X} \in \mathbb{R}^{n \cdot m}_{\geq 0}$. ◁

Baruah and Brandenburg (2013) and Voronov and Anderson (2018) presented optimal table-driven schedulers for any IDENTICAL/ARBITRARY-Feasible task systems.

Two EDF variants for IDENTICAL/ARBITRARY targeting HRT that are of relevance to this dissertation are Strong-APA-EDF and Weak-APA-EDF (Cerqueira et al., 2014). Defining these variants requires the concept of shifting.

▽ **Definition 2.28.** *Shifting* (Cerqueira et al., 2014) is the series of migrations that results from inverting each edge in an alternating path originating with an unmatched (*i.e.*, unscheduled) task. △

▷ **Weak-APA-EDF (Cerqueira et al., 2014).** At any time, the chosen configuration is such that no unscheduled ready task τ_i should have affinity for a processor that schedules either no job or a job $\tau_{k,\ell}$ such that $d_i(t) < d_{k,\ell}$. ◁

[7]Note that offline partitioning is optional for EDF-sc.

(a) Weak-APA-EDF. (b) Strong-APA-EDF.

Figure 2.7: Weak-APA-EDF versus Strong-APA-EDF.

▼ **Example 2.8.** Consider a system of $n = 5$ tasks and $m = 3$ processors with affinities as illustrated in the graphs of Figure 2.7. Suppose that at some time t, all tasks have ready jobs such that $d_2(t) < d_4(t) < d_5(t) < d_1(t) < d_3(t)$. Consider the configuration illustrated in Figure 2.7a. Because unscheduled task τ_5 (resp., τ_3) cannot preempt τ_4 (resp., τ_2) on π_3 (resp., π_2) as $d_4(t) < d_5(t)$ (resp., $d_2(t) < d_3(t)$), this configuration is possible at time t under Weak-APA-EDF. Likewise, because tasks τ_1 and τ_3 cannot preempt higher-priority tasks in Figure 2.7b, this configuration is also possible under Weak-APA-EDF.▲

▷ **Strong-APA-EDF (Cerqueira et al., 2014).** At any time, the chosen configuration is such that no unscheduled ready task τ_i has an alternating path beginning at τ_i and ending with either a processor that schedules no job or a job $\tau_{k,\ell}$ such that $d_i(t) < d_{k,\ell}$. ◁

▼ **Example 2.9.** Recall the system detailed in Example 2.8. Observe in Figure 2.7a that an alternating path $\{\tau_5, \pi_3, \tau_4, \pi_2, \tau_2, \pi_1, \tau_1\}$ can be traced from the higher-priority (job of) τ_5 to the lower-priority (job of) τ_1. Thus, the configuration in Figure 2.7a is impossible under Strong-APA-EDF. By inverting the edges in this path (*i.e.*, shifting), one arrives at Figure 2.7b. Because no shifts can result in the scheduling of a higher-priority task in Figure 2.7b, this configuration is possible under Strong-APA-EDF. ▲

Weak-APA-EDF is of significance because it describes the behavior of most EDF implementations under affinities, such as `SCHED_DEADLINE`.[8] Strong-APA-EDF is of significance because we will prove that it is SRT-optimal under IDENTICAL/ARBITRARY in Chapter 3.

Cerqueira et al. (2014) demonstrated that computing configurations under Strong-APA-EDF can be done via solving MVM instances where edge set $\mathbb{E}$ reflects the affinity graph. This results in the alternative definition of Strong-APA-EDF below.

[8]With some exceptions, detailed in Chapter 4.

▷ **Strong-APA-EDF—MVM definition (Cerqueira et al., 2014).** At any time t, the chosen configuration corresponds with an optimal solution of $\text{MVM}\left(\tau, \pi, \overrightarrow{\psi}, \mathbb{E}\right)$ such that

$$\mathbb{E} = \{(\tau_i, \pi_j) : \tau_i \in \tau \text{ and } \pi_j \in \alpha_i\}, \tag{2.11}$$

i.e., the edges reflect tasks' affinities,

$$\forall \tau_i \in \tau : \tau_i \in \tau_{\text{rdy}}(t) \Rightarrow \psi_i > 0 \text{ and } \tau_i \notin \tau_{\text{rdy}}(t) \Rightarrow \psi_i = 0, \tag{2.12}$$

i.e., ready tasks have positive weight and non-ready tasks have 0 weight, and

$$\forall \tau_i, \tau_j \in \tau_{\text{rdy}}(t) : d_i(t) < d_j(t) \Rightarrow \psi_i > \psi_j, \tag{2.13}$$

i.e., tasks with earlier deadlines have higher weight. ◁

Note that for any time t, there is always a canonical configuration that optimally solves the corresponding MVM instance because any optimal solution $\mathbf{X}$ can be transformed into a canonical configuration without changing the objective function value. Suppose we have task τ_i and processor π_j such that τ_i is non-ready and $x_{i,j} = 1$. By (2.12), a task τ_i has $\psi_i = 0$. Because $x_{i,j}$ is multiplied by ψ_i in (2.5), setting $x_{i,j} = 0$ does not change the objective function value. Likewise, suppose we have task τ_i and processor π_j such that $\pi_j \notin \alpha_i$ and $x_{i,j} = 1$. By (2.11), $(\tau_i, \pi_j) \notin \mathbb{E}$. Because $x_{i,j}$ is only present in (2.5) if $(\tau_i, \pi_j) \in \mathbb{E}$, setting $x_{i,j} = 0$ does not change the objective function value. By Definition 2.26, $\mathbf{X}$ is canonical after setting each such $x_{i,j}$ to 0.

For such MVM instances corresponding with Strong-APA-EDF, outside of (2.13), Cerqueira et al. (2014) did not specify how ψ_i should be specified for task τ_i (their work primarily considered Weak APA and Strong APA variants of fixed-priority schedulers, under which ψ_i was defined as task τ_i's priority level).

They further proved that under Strong-APA-EDF, a single job $\tau_{i,j}$ completion belonging to task τ_i (thus increasing $d_i(t)$ and potentially making τ_i non-ready) requires at most one shift terminating with τ_i (or $\tau_{i,j}$'s former processor if τ_i becomes non-ready) to return to a Strong-APA-EDF configuration. Likewise, a single job $\tau_{i,j}$ arrival (potentially making τ_i ready) requires at most one shift originating with τ_i. Regardless of whether job $\tau_{i,j}$ completes or arrives, the required shift can be computed in $O(n \cdot m)$ time using a

breadth-first-search on the affinity graph (starting with task τ_i). This is more efficient than solving a new MVM instance from scratch.

2.4.3 Work under UNIFORM

Prior work has established the SRT-optimality of EDF under UNIFORM. The feasibility condition under UNIFORM is as follows.

▷ **UNIFORM-Feasbility (Funk et al., 2001).** Task system τ is feasible if and only if

$$\forall k \in \{1, 2, \ldots, |\tau|\} : \sum_{i=1}^{k} u_{[i]} \leq \sum_{j=1}^{\min\{m,k\}} sp^{([j])}. \qquad \triangleleft$$

Yang and Anderson (2015) initially disproved that any sporadic UNIFORM-Feasible task system with non-preemptive jobs has bounded response times via a counterexample. In fact, this counterexample showed that any work-conserving scheduler (*i.e.*, one that never leaves a task unscheduled when processors exists that are not executing other jobs) may have unbounded response times, even if the system is UNIFORM-Feasible. This suggests that the SRT-optimality of non-preemptive EDF is unique to IDENTICAL.

Yang and Anderson (2017) went on to prove that under an EDF variant that we denote as Ufm-EDF, for any sporadic UNIFORM-Feasible task system with fully-preemptive jobs, task τ_i has response time at most

$$T_i + \frac{\left(u_{[1]}/u_{[n]}\right)^{m-1}(n-m+1)\,C_{[1]} + \frac{\left(u_{[1]}/u_{[n]}\right)^{m-1}-1}{\left(u_{[1]}/u_{[n]}\right)-1} C_{[1]}}{u_i}.$$

This response-time bound is exponential in m. Ufm-EDF is defined as follows.

▷ **Ufm-EDF (Yang and Anderson, 2017).** At any time t, the ready task with earliest deadline is scheduled on the fastest processor, the ready task with second earliest deadline on the second fastest processor, and so on until all ready tasks are scheduled or all processors are scheduled upon. $\triangleleft$

Yang and Anderson (2014) proved response-time bounds for NPC-sporadic tasks under Ufm-EDF. Assuming implicit deadlines, these bounds are $O\left(\frac{m \cdot C_{[1]}}{\sum_{\pi_j \in \pi} sp^{(j)}}\right)$. Yang and Anderson (2014) also proved $O\left(\frac{m \cdot C_{[1]}}{\sum_{\pi_j \in \pi} sp^{(j)}}\right)$ response-time bounds for any UNIFORM-Feasible NPC-sporadic task system under non-preemptive Ufm-EDF. Note that for NPC-sporadic tasks, due to the allowance of intra-task parallelism, the condition $\sum_{\tau_i \in \tau} u_i \leq \sum_{\pi_j \in \pi} sp^{(j)}$ is necessary and sufficient for UNIFORM-Feasible.

2.4.4 Work under UNRELATED

Perhaps due to their complexity, UNRELATED multiprocessors are less frequently considered in the real-time literature. The feasibility condition under UNRELATED is as follows.

▷ **UNRELATED-Feasibility (Baruah, 2004).** Task system τ is feasible if and only if

$$\forall \tau_i \in \tau : \sum_{\pi_j \in \pi} sp^{i,j} \cdot x_{i,j} \geq u_i, \tag{2.14}$$

$$\forall \tau_i \in \tau : \sum_{\pi_j \in \pi} x_{i,j} \leq 1.0, \tag{2.15}$$

$$\forall \pi_j \in \pi : \sum_{\tau_i \in \tau} x_{i,j} \leq 1.0, \tag{2.16}$$

for some $\mathbf{X} \in \mathbb{R}^{n \cdot m}_{\geq 0}$. ◁

Though the author is unaware of prior works explicitly considering SRT analysis under UNRELATED multiprocessors, existing table-driven schedulers targeting HRT can be made SRT by increasing table iteration lengths. Baruah (2004), as part of proving the above UNRELATED-Feasible condition, presented an algorithm for generating a table-driven schedule that meets all deadlines for any UNRELATED-Feasible task system. Chwa et al. (2015) presented a table generation algorithm for 2-type UNRELATED multiprocessors with lower time complexity.

Instead of allowing tasks to migrate freely, the more common approach for real-time scheduling on UNRELATED multiprocessors seems to be enforcing either PARTITIONED affinities or, for a k-type multiprocessor, CLUSTERED affinities with a cluster corresponding to each type. This reduces analytical complexity because under PARTITIONED affinities, each processor can be analyzed as if it were an independent uniprocessor (*e.g.*, for sporadic task τ_i with affinity for a single processor π_j, we can rescale C_i to $C_i \leftarrow C_i / sp^{i,j}$ to make π_j appear as a unit-speed processor). Likewise, under CLUSTERED affinities with a cluster per type, each cluster can be analyzed as if it were an independent IDENTICAL multiprocessor. The problem of interest then becomes how to intelligently partition the set of tasks among the processors or clusters, which becomes conceptually equivalent to various flavors of bin-packing. As a result, this partitioning problem is typically solved using heuristics or approximation algorithms. Baruah et al. (2016) presented several ILPs for partitioning tasks for PARTITIONED affinities, after which tasks are scheduled as in uniprocessor EDF. Because bins are rarely completely filled in bin-packing, the analogue for the task

partitioning problem is that some processors will inevitably be underloaded. This wastes processing capacity. As such, this lost processing capacity means the partitioning approach cannot yield optimality (*i.e.*, timing guarantees can be met for any UNRELATED-Feasible system).

2.5 Chapter Summary

In this chapter, we covered the sporadic task model and CBS. We defined the WC class of schedulers. We reviewed relevant optimization problems and related work.

CHAPTER 3: RESPONSE-TIME BOUNDS[1]

In this chapter, we prove response-time bounds for WC schedulers on various multiprocessor models.

3.1 Deviation

This section proves properties about a function we call *deviation*, a measure of how behind a task's execution is at a specific time instant. Deviation is similar to the well-known concept of lag, but is more closely tied to deadlines than lag when arrivals are sporadic and jobs do not execute to their worst-case execution times. We abstract the state of the task system using deviation. This is because deviation is (mostly) continuous, which makes deviation simpler to reason about than the unabstracted task system state.

The definition of deviation relies on that of virtual time.

▽ **Definition 3.1.** The *virtual time* of task τ_i is

$$vt_i(t) \triangleq a_i(t) + T_i \frac{c_i(t) - rem_i(t)}{c_i(t)}. \qquad \triangle$$

▽ **Definition 3.2.** The *deviation* of task τ_i at time t is

$$dev_i(t) \triangleq \begin{cases} \sqrt{u_i} \cdot (t - vt_i(t)) & t \geq vt_i(t) \\ 0 & t < vt_i(t) \end{cases}. \qquad \triangle$$

We now prove properties of virtual time and deviation used in the remainder of this chapter.

[1]Contents of this chapter previously appeared in the following papers:

Stephen Tang, Sergey Voronov, and James H. Anderson. GEDF tardiness: Open problems involving uniform multiprocessors and affinity masks resolved. In *31st Euromicro Conference on Real-Time Systems*, volume 133, pages 13:1–13:21, 2019.

Stephen Tang and James H. Anderson. Towards practical multiprocessor EDF with affinities. In *41st IEEE Real-Time Systems Symposium*, pages 89–101, 2020.

Stephen Tang, Sergey Voronov, and James H. Anderson. Extending EDF for soft real-time scheduling on unrelated multiprocessors. In *2021 IEEE Real-Time Systems Symposium*, pages 253–265, 2021b.

3.1.1 Scheduling

Lemmas proven in this subsection relate a task's deviation to whether said task is ready and to said task's priority point.

▷ **Lemma 3.1.** For task τ_i and time t, if we have $dev_i(t) > 0$, then task τ_i is ready at t. ◁

Proof. Let $\tau_{i,j}$ denote the current job of task τ_i at time t. We have

$$
\begin{aligned}
t > & \{dev_i(t) > 0 \text{ and Definition 3.2}\} \\
& vt_i(t) \\
= & \{\text{Definition 3.1}\} \\
& a_i(t) + T_i \frac{c_i(t) - rem_i(t)}{c_i(t)} \\
\geq & \{rem_i(t) \leq c_i(t)\} \\
& a_i(t) \\
= & \{\text{Definition 2.10}\} \\
& a_{i,j} \\
\geq & \{\text{Definition 2.11}\} \\
& rdy_{i,j}.
\end{aligned}
$$

By Definition 2.11, task τ_i is ready at time t. □

▷ **Lemma 3.2.** For any task τ_i, we have $vt_i(t) < \tilde{d}_i(t) \leq vt_i(t) + T_i$. ◁

Proof. By Definition 3.1, we have $vt_i(t) = a_i(t) + T_i \frac{c_i(t) - rem_i(t)}{c_i(t)}$. Because $rem_i(t) > 0$ (recall that, by Definition 2.10, the current job is incomplete by definition, thus, the remaining execution of the current job $rem_i(t)$ cannot be 0) and $rem_i(t) \leq c_i(t)$, we have $a_i(t) \leq vt_i(t) < a_i(t) + T_i$. Because, by Definition 2.5, $\tilde{d}_i(t) = a_i(t) + T_i$, we have $\tilde{d}_i(t) - T_i \leq vt_i(t) < \tilde{d}_i(t)$. Rearrangement yields the lemma statement. □

▷ **Lemma 3.3.** Consider an arbitrary WC scheduler. If at time t, we have $\frac{dev_e(t)}{\sqrt{u_e}} \geq \frac{dev_\ell(t)}{\sqrt{u_\ell}} + T_{[1]} + 2\phi$, then $pp_e(t) < pp_\ell(t)$. ◁

Proof. We start by proving the following claim.

► **Claim 3.3.1.** $t > vt_e(t)$. ◄

Proof. We have

$$
\begin{aligned}
0 < & \ \{T_{[1]} > 0 \text{ and, by Definition 2.20, } \phi \geq 0\} \\
& \ T_{[1]} + 2\phi \\
\leq & \ \{\text{By Definition 3.2, } dev_\ell(t) \geq 0\} \\
& \ \frac{dev_\ell(t)}{\sqrt{u_\ell}} + T_{[1]} + 2\phi \\
\leq & \ \{\text{Lemma statement}\} \\
& \ \frac{dev_e(t)}{\sqrt{u_e}}.
\end{aligned}
$$

Thus, we have $dev_e(t) > 0$. The claim follows from Definition 3.2. ■

Claim 3.3.1 is used in the following derivation to show that $pp_e(t) < pp_\ell(t)$, which is our proof obligation.

$$
\begin{aligned}
pp_e(t) \leq & \ \{\text{Definition 2.20}\} \\
& \ \tilde{d}_e(t) + \phi \\
\leq & \ \{\text{Lemma 3.2}\} \\
& \ vt_e(t) + T_e + \phi \\
\leq & \ \{T_e \leq T_{[1]}\} \\
& \ vt_e(t) + T_{[1]} + \phi \\
= & \ -(t - vt_e(t)) + t + T_{[1]} + \phi \\
= & \ \{\text{Claim 3.3.1 and Definition 3.2}\} \\
& \ -\frac{dev_e(t)}{\sqrt{u_e}} + t + T_{[1]} + \phi \\
\leq & \ \{\text{Lemma statement}\} \\
& \ -\frac{dev_\ell(t)}{\sqrt{u_\ell}} - T_{[1]} - 2\phi + t + T_{[1]} + \phi
\end{aligned}
$$

$$
\begin{aligned}
&= -\frac{dev_\ell(t)}{\sqrt{u_\ell}} - \phi + t \\
&\leq \left\{\text{By Definition 3.2, } \frac{dev_\ell(t)}{\sqrt{u_\ell}} \geq t - vt_\ell(t)\right\} \\
&\quad -(t - vt_\ell(t)) - \phi + t \\
&= vt_\ell(t) - \phi \\
&< \{\text{Lemma 3.2}\} \\
&\quad \tilde{d}_\ell(t) - \phi \\
&\leq \{\text{Definition 2.20}\} \\
&\quad pp_\ell(t)
\end{aligned}
$$

□

3.1.2 Response Times

Lemmas proven in this subsection relate a task's deviation to its response times.

▷ **Lemma 3.4.** If, for some $\ell > 0$, for all time instants t, we have $dev_i(t) \leq \ell \cdot \sqrt{u_i}$, then the response time of any job $\tau_{i,j}$ is at most $T_i + \ell$. ◁

Proof. We prove the contrapositive: if the response time of any job $\tau_{i,j}$ exceeds $T_i + \ell$, then for some time instant t we have $dev_i(t) > \ell \cdot \sqrt{u_i}$. Let job $\tau_{i,j}$ be a job with response time exceeding $T_i + \ell$, *i.e.*, $f_{i,j} - a_{i,j} > T_i + \ell$.

Let time $t^* \triangleq \tilde{d}_{i,j} + \ell$. By Definition 2.5, we have $t^* = T_i + a_{i,j} + \ell$. Because $f_{i,j} - a_{i,j} > T_i + \ell$, we have $f_{i,j} > t^*$, meaning job $\tau_{i,j}$ is incomplete at time t^*. Thus, either job $\tau_{i,j}$ or an earlier job of task τ_i must be the current job of task τ_i at t^* (recall that, by Definition 2.10, the current job of a task is the incomplete job of said task with the earliest arrival time), so $\tilde{d}_i(t^*) \leq \tilde{d}_{i,j}$. Because $t^* = \tilde{d}_{i,j} + \ell$, we have $t^* \geq \tilde{d}_i(t^*) + \ell \Rightarrow t^* - vt_i(t^*) \geq \tilde{d}_i(t^*) - vt_i(t^*) + \ell$. By Lemma 3.2, $\tilde{d}_i(t^*) > vt_i(t^*)$. Because $t^* - vt_i(t^*) \geq \tilde{d}_i(t^*) - vt_i(t^*) + \ell$, we have $t^* - vt_i(t^*) > \ell$. By Definition 3.2, we have $dev_i(t^*) > \ell \cdot \sqrt{u_i}$. □

▷ **Lemma 3.5.** For task τ_i and time t, if τ_i is inactive, then $dev_i(t) = 0$. ◁

Proof. By Definition 2.13, we have that task τ_i is not ready at time t. Let job $\tau_{i,j}$ be task τ_i's current job. By Definition 2.11, we have that $t < rdy_{i,j}$. Because $rdy_{i,j} \leq a_{i,j}$ (also by Definition 2.11), we have

$t < a_{i,j}$. Subtracting $vt_i(t)$ from both sides, we have

$$\begin{aligned} t - vt_i(t) &< a_{i,j} - vt_i(t) \\ &= \{\text{Definition 2.10}\} \\ &\quad a_i(t) - vt_i(t) \\ &\leq \{rem_i(t) \leq c_i(t)\} \\ &\quad a_i(t) + T_i \frac{c_i(t) - rem_i(t)}{c_i(t)} - vt_i(t) \\ &= \{\text{Definition 3.1}\} \\ &\quad vt_i(t) - vt_i(t) \\ &= 0. \end{aligned}$$

The lemma follows from Definition 3.2. □

3.1.3 Evolution

Lemmas proven in this subsection relate to how deviation changes over time.

▷ **Lemma 3.6.** For task τ_i, time instant t, and $\epsilon > 0$, we have $vt_i(t + \epsilon) \geq vt_i(t)$. ◁

Proof. Let jobs $\tau_{i,j}$ and τ_{i,j^*} denote the current job of task τ_i at times t and $t + \epsilon$, respectively. There are two cases to consider.

◀ **Case 3.6.1.** The current job of task τ_i is the same at times t and $t + \epsilon$, *i.e.*, $j = j^*$. ▶

By Definition 2.10, we have

$$a_i(t) = a_i(t + \epsilon) \text{ and } c_i(t) = c_i(t + \epsilon). \tag{3.1}$$

Because $\tau_{i,j}$ and τ_{i,j^*} are the same job and the remaining execution required by a job does not increase over time, we have

$$rem_i(t + \epsilon) \leq rem_i(t). \tag{3.2}$$

Thus,

$$\begin{aligned} vt_i(t+\epsilon) &= \{\text{Definition 3.1}\} \\ &\quad a_i(t+\epsilon) + T_i \frac{c_i(t+\epsilon) - rem_i(t+\epsilon)}{c_i(t+\epsilon)} \\ &= \{\text{Equation (3.1)}\} \\ &\quad a_i(t) + T_i \frac{c_i(t) - rem_i(t+\epsilon)}{c_i(t)} \\ &\geq \{\text{Equation (3.2)}\} \\ &\quad a_i(t) + T_i \frac{c_i(t) - rem_i(t)}{c_i(t)} \\ &= \{\text{Definition 3.1}\} \\ &\quad vt_i(t). \end{aligned}$$ ♦

◄ **Case 3.6.2.** The current jobs of task τ_i at times t and $t+\epsilon$ are distinct, *i.e.*, $j \neq j^*$. ►

Because the index of the current job never decreases, we have $j^* > j$. Because the remaining cost of the current job is at most its total cost, we have

$$rem_i(t+\epsilon) \leq c_i(t+\epsilon) \text{ and } rem_i(t) \leq c_i(t) \tag{3.3}$$

Thus,

$$\begin{aligned} vt_i(t+\epsilon) &= \{\text{Definition 3.1}\} \\ &\quad a_i(t+\epsilon) + T_i \frac{c_i(t+\epsilon) - rem_i(t+\epsilon)}{c_i(t+\epsilon)} \\ &\geq \{\text{Equation (3.3)}\} \\ &\quad a_i(t+\epsilon) \\ &= \{\text{Definition 2.10}\} \\ &\quad a_{i,j^*} \\ &\geq \{j^* > j \text{ and task } \tau_i \text{ is a sporadic task}\} \\ &\quad a_{i,j} + T_i \end{aligned}$$

$$
\begin{aligned}
&> \{\text{By Definition 2.10, } c_i(t) > 0 \text{ and } rem_i(t) > 0\} \\
&\quad a_{i,j} + T_i \frac{c_i(t) - rem_i(t)}{c_i(t)} \\
&= \{\text{Definition 3.1}\} \\
&\quad vt_i(t).
\end{aligned}
$$

♦

For both cases, $vt_i(t+\epsilon) \geq vt_i(t)$. □

▷ **Lemma 3.7.** For any task τ_i and time t, there exists $\psi > 0$ such that both

$$\forall t^* \in [t, t+\psi) : \ vt_i(t^*) = vt_i(t) + (t^* - t) \frac{T_i}{c_i(t)} csp_i(t) \tag{3.4}$$

and

$$\forall t^* \in [t-\psi, t) : \ vt_i(t^*) = vt_i(t-\psi) + (t^* - t + \psi) \frac{T_i}{c_i(t-\psi)} csp_i(t-\psi). \tag{3.5}$$

are true. ◁

Proof. We consider proving (3.4) first. We select arbitrarily small ψ such that propositions (3.6) and (3.7), defined below, are true.

Let job $\tau_{i,j}$ be the current job of task τ_i at time t. For small enough ψ, because of the Non-Fluid Assumption, job $\tau_{i,j}$ is still the current job of task τ_i throughout $[t, t+\psi)$. By Definition 2.10,

$$\forall t^* \in [t, t+\psi) : \ a_i(t^*) = a_i(t) \text{ and } c_i(t^*) = c_i(t). \tag{3.6}$$

By the Non-Fluid Assumption, for small enough ψ, task τ_i is, over the interval $[t, t+\psi)$, either scheduled on the same processor or not scheduled on any processor. By Definition 2.21, we have $\forall t^* \in [t, t+\psi) : \ csp_i(t^*) = csp_i(t)$. Because the current job of task τ_i is $\tau_{i,j}$ over $[t, t+\psi)$, we have

$$\forall t^* \in [t, t+\psi), rem_i(t^*) = rem_i(t) - (t^* - t) \cdot csp_i(t), \tag{3.7}$$

i.e., the remaining execution of the current job is decreased by the duration of execution $(t^* - t)$ multiplied by the execution speed $csp_i(t)$.

We have

$$\left.\begin{aligned} vt_i(t^*) &= \{\text{Definition 3.1}\} \\ &\quad a_i(t^*) + T_i \frac{c_i(t^*) - rem_i(t^*)}{c_i(t^*)} \\ &= \{\text{Equation (3.6)}\} \\ &\quad a_i(t) + T_i \frac{c_i(t) - rem_i(t^*)}{c_i(t)} \\ &= \{\text{Equation (3.7)}\} \\ &\quad a_i(t) + T_i \frac{c_i(t) - rem_i(t) + (t^* - t) \cdot csp_i(t)}{c_i(t)} \\ &= a_i(t) + T_i \frac{c_i(t) - rem_i(t)}{c_i(t)} + (t^* - t) \frac{T_i}{c_i(t)} csp_i(t) \\ &= \{\text{Definition 3.1}\} \\ &\quad vt_i(t) + (t^* - t) \frac{T_i}{c_i(t)} csp_i(t). \end{aligned}\right\} \tag{3.8}$$

(3.8) is (3.4), which is half of our proof obligations.

The reasoning for (3.5) is similar. By the Non-Fluid Assumption, for small enough ψ, we have that the same job $\tau_{i,j}$ is the current job of task τ_i over $[t - \psi, t)$. Additionally, task τ_i is either scheduled on the same processor or is unscheduled over $[t - \psi, t)$. (3.5) is yielded by substituting all instances of t with $t - \psi$ in (3.6)-(3.8). □

▷ **Lemma 3.8.** For task τ_i and time t, the left-sided limit $\lim_{t^* \to t^-} vt_i(t^*)$ is well-defined. ◁

Proof. Our proof obligation is to show that $vt_i(t^*)$ approaches some finite value ℓ as $t^* \to t^-$, *i.e.*,

$$\exists \ell \in \mathbb{R} : \ \forall \epsilon > 0 : \ \exists : \delta > 0 : \ \forall t^* \in (t - \delta, t) : \ |vt_i(t^*) - \ell| \leq \epsilon. \tag{3.9}$$

By (3.5) of Lemma 3.7, we have

$$\exists \psi > 0 : \ \forall t^* \in [t - \psi, t) : \ vt_i(t^*) = vt_i(t - \psi) + (t^* - (t - \psi)) \frac{T_i}{c_i(t - \psi)} csp_i(t - \psi). \tag{3.10}$$

We consider two cases depending on the value of $csp_i(t - \psi)$. Note that, as a processor speed, this value is non-negative.

◀ **Case 3.8.1.** $csp_i(t - \psi) = 0$. ▶

Let $\ell \triangleq vt_i(t-\psi)$. Consider any $\epsilon > 0$. For any $t^* \in [t-\psi, t)$, we have

$$\begin{aligned}
vt_i(t^*) - \ell = & \{\text{Equation (3.10)}\} \\
& vt_i(t-\psi) + (t^* - (t-\psi))\,\frac{T_i}{c_{i,j}}\,csp_i(t-\psi) - \ell \\
= & \{csp_i(t-\psi) = 0\} \\
& vt_i(t-\psi) + 0 - \ell \\
= & \{\text{Definition of } \ell\} \\
& vt_i(t-\psi) - vt_i(t-\psi) \\
= & 0 \\
< & \epsilon.
\end{aligned}$$

This implies (3.9), the proof obligation. ♦

◄ **Case 3.8.2.** $csp_i(t-\psi) > 0$. ►

Let

$$\ell \triangleq vt_i(t-\psi) + \psi\frac{T_i}{c_i(t-\psi)}\,csp_i(t-\psi). \tag{3.11}$$

For any $\epsilon > 0$, let

$$\delta \triangleq \min\left\{\psi, \epsilon \cdot \frac{c_i(t-\psi)}{T_i \cdot csp_i(t-\psi)}\right\}. \tag{3.12}$$

We have $\forall t^* \in (t-\delta, t)$:

$$\begin{aligned}
|vt_i(t^*) - \ell| = & \{\text{Equation (3.10)}\} \\
& \left|vt_i(t-\psi) + (t^* - (t-\psi))\,\frac{T_i}{c_i(t-\psi)}\,csp_i(t-\psi) - \ell\right| \\
= & \left|vt_i(t-\psi) + \psi\frac{T_i}{c_i(t-\psi)}\,csp_i(t-\psi) + (t^* - t)\,\frac{T_i}{c_i(t-\psi)}\,csp_i(t-\psi) - \ell\right| \\
= & \{\text{Equation (3.11)}\} \\
& \left|(t^* - t)\,\frac{T_i}{c_i(t-\psi)}\,csp_i(t-\psi)\right|
\end{aligned}$$

$$
\begin{aligned}
&= \{t^* < t\} \\
&\quad (t - t^*)\,\frac{T_i}{c_i(t-\psi)}\,csp_i(t-\psi) \\
&< \{t^* > t - \delta\} \\
&\quad \delta\frac{T_i}{c_i(t-\psi)}\,csp_i(t-\psi) \\
&\leq \left\{\text{By Equation (3.12), } \delta \leq \epsilon \cdot \frac{c_i(t-\psi)}{T_i \cdot csp_i(t-\psi)}\right\} \\
&\quad \epsilon.
\end{aligned}
$$

This implies (3.9), the proof obligation. ♦

Both cases yield (3.9), the proof obligation. □

The following corollary follows from Lemma 3.8 and Definition 3.2.

▷ **Corollary 3.9.** For task τ_i and time t, the left-sided limit $\lim_{t^* \to t^-} dev_i(t^*)$ is well-defined. ◁

Proof.

$$
\begin{aligned}
\lim_{t^* \to t^-} dev_i(t^*) &= \{\text{Definition 3.2}\} \\
&\quad \lim_{t^* \to t^-} \max\{0, \sqrt{u_i} \cdot (t^* - vt_i(t^*))\} \\
&= \max\left\{\lim_{t^* \to t^-} 0, \lim_{t^* \to t^-} \sqrt{u_i} \cdot (t^* - vt_i(t^*))\right\} \\
&= \max\left\{0, \sqrt{u_i} \cdot \left(\lim_{t^* \to t^-} (t^* - vt_i(t^*))\right)\right\} \\
&= \max\left\{0, \sqrt{u_i} \cdot \left(\left[\lim_{t^* \to t^-} t^*\right] - \left[\lim_{t^* \to t^-} vt_i(t^*)\right]\right)\right\} \\
&= \max\left\{0, \sqrt{u_i} \cdot \left(t - \left[\lim_{t^* \to t^-} vt_i(t^*)\right]\right)\right\}
\end{aligned}
$$

By Lemma 3.8, this value is well-defined. □

Corollary 3.9 is necessary for Lemma 3.10 to be well-defined.

▷ **Lemma 3.10.** For task τ_i and time t, we have

$$\lim_{t^* \to t^-} dev_i(t^*) \geq dev_i(t).$$ ◁

Proof. We have

$$\begin{aligned}
\lim_{t^* \to t^-} dev_i(t^*) &= \{\text{Definition 3.2}\} \\
&\quad \lim_{t^* \to t^-} \max\{0, \sqrt{u_i} \cdot (t^* - vt_i(t^*))\} \\
&= \max\left\{0, \sqrt{u_i} \cdot \lim_{t^* \to t^-} (t^* - vt_i(t^*))\right\} \\
&= \max\left\{0, \sqrt{u_i} \cdot \left(t - \lim_{t^* \to t^-} vt_i(t^*)\right)\right\} \\
&\geq \{\text{Lemma 3.6}\} \\
&\quad \max\{0, \sqrt{u_i} \cdot (t - vt_i(t))\} \\
&= \{\text{Definition 3.2}\} \\
&\quad dev_i(t).
\end{aligned}$$

□

▷ **Lemma 3.11.** For task τ_i and time t, the right-sided limit $\lim_{t^* \to t^+} vt_i(t^*) = vt_i(t)$. ◁

Proof. By (3.4) of Lemma 3.7, there exists $\psi > 0$ such that $\forall t^* \in [t, t + \psi) : \ vt_i(t^*) = vt_i(t) + (t^* - t)\frac{T_i}{c_i(t)} csp_i(t)$. Thus,

$$\begin{aligned}
\lim_{t^* \to t^+} vt_i(t^*) &= \lim_{t^* \to t^+} \left(vt_i(t) + (t^* - t)\frac{T_i}{c_i(t)} csp_i(t)\right) \\
&= vt_i(t) + (0)\,\frac{T_i}{c_i(t)} csp_i(t) \\
&= vt_i(t).
\end{aligned}$$

□

The following corollary follows from Lemma 3.11 and Definition 3.2.

▷ **Corollary 3.12.** For task τ_i and time t, the right-sided limit $\lim_{t^* \to t^+} dev_i(t^*) = dev_i(t)$. ◁

Proof.

$$\begin{aligned}
\lim_{t^* \to t^+} dev_i(t^*) &= \{\text{Definition 3.2}\} \\
&\quad \lim_{t^* \to t^+} \max\{0, \sqrt{u_i} \cdot (t^* - vt_i(t^*))\} \\
&= \max\left\{0, \sqrt{u_i} \cdot \left(\left\lfloor \lim_{t^* \to t^+} t^* \right\rfloor - \left\lfloor \lim_{t^* \to t^+} vt_i(t^*) \right\rfloor\right)\right\}
\end{aligned}$$

$$
\begin{aligned}
&= \{\text{Lemma 3.11}\} \\
&\quad \max\{0, \sqrt{u_i} \cdot (t - vt_i(t))\} \\
&= \{\text{Definition 3.2}\} \\
&\quad dev_i(t)
\end{aligned}
$$

□

▷ **Lemma 3.13.** For any task τ_i and time t such that $dev_i(t) > 0$, there exists $\psi > 0$ such that

$$\forall t^* \in [t, t+\psi) : \ \sqrt{u_i} \cdot dev_i(t^*) \leq \sqrt{u_i} \cdot dev_i(t) + (t^* - t) \cdot (u_i - csp_i(t)) \,. \qquad \triangleleft$$

Proof. By Lemma 3.7, we can select $\psi > 0$ small enough such that (3.4) is true. Thus, for $t^* \in [t, t+\psi)$, we have

$$
\begin{aligned}
&dev_i(t^*) \\
&= \{\text{Definition 3.2}\} \\
&\quad \max\{0, \sqrt{u_i} \cdot (t^* - vt_i(t^*))\} \\
&= \{\text{Equation (3.4)}\} \\
&\quad \max\left\{0, \sqrt{u_i} \cdot \left(t^* - vt_i(t) - (t^* - t)\frac{T_i}{c_i(t)} csp_i(t)\right)\right\} \\
&= \max\left\{0, \sqrt{u_i} \cdot (t - vt_i(t)) + (t^* - t)\sqrt{u_i}\left(1 - \frac{T_i}{c_i(t)} csp_i(t)\right)\right\} \\
&= \{\text{Definition 3.2 and } dev_i(t) > 0\} \\
&\quad \max\left\{0, dev_i(t) + (t^* - t)\sqrt{u_i}\left(1 - \frac{T_i}{c_i(t)} csp_i(t)\right)\right\} \\
&\leq \left\{-\frac{T_i}{c_i(t)} \leq -\frac{T_i}{C_i} = -\frac{1}{u_i}\right\} \\
&\quad \max\left\{0, dev_i(t) + (t^* - t)\sqrt{u_i}\left(1 - \frac{csp_i(t)}{u_i}\right)\right\} \\
&= \max\left\{0, dev_i(t) + (t^* - t) \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}}\right)\right\}.
\end{aligned}
$$

There are two cases depending on the value of $\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}}$.

◄ **Case 3.13.1.** $\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}} \geq 0$. ►

$$
\begin{aligned}
dev_i(t^*) &\leq \max\left\{0, dev_i(t) + (t^* - t) \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}}\right)\right\} \\
&\leq \left\{dev_i(t) > 0, t^* \geq t, \text{and } \sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}} \geq 0\right\} \\
&\quad dev_i(t) + (t^* - t) \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}}\right)
\end{aligned}
$$

The lemma follows from multiplying by $\sqrt{u_i}$. ♦

◄ **Case 3.13.2.** $\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}} < 0$. ►

Let

$$
\psi' \triangleq \min\left\{\psi, \frac{dev_i(t)}{\frac{csp_i(t)}{\sqrt{u_i}} - \sqrt{u_i}}\right\}.
$$

Because $\psi' \leq \psi$, we have $\forall t^* \in [t, t + \psi')$:

$$
\begin{aligned}
dev_i(t^*) &\leq \max\left\{0, dev_i(t) + (t^* - t) \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}}\right)\right\} \\
&= \left\{(t^* - t) < \psi' \leq \frac{dev_i(t)}{\frac{csp_i(t)}{\sqrt{u_i}} - \sqrt{u_i}} \Rightarrow dev_i(t) + (t^* - t) \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}}\right) \geq 0\right\} \\
&\quad dev_i(t) + (t^* - t) \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}}\right)
\end{aligned}
$$

The lemma follows from multiplying by $\sqrt{u_i}$. ♦

Either case yields the lemma. □

3.1.4 Proof Strategy

Proofs of response-time bounds for EDF variants under the considered multiprocessor models (UNIFORM, IDENTICAL/ARBITRARY, and UNRELATED) all follow the same general strategy, which we give a high-level overview of here. This strategy shows that vector $\overrightarrow{\mathbf{dev}}(t) = \langle dev_1(t), dev_2(t), \ldots, dev_n(t)\rangle$ remains in a bounded region of $\mathbb{R}^n_{\geq 0}$ (recall that, by Definition 3.2, $dev_i(t) \geq 0$) for any time t. This implies

that, for each task τ_i, we have that $dev_i(t)$ is upper bounded for any time t. Thus, by Lemma 3.4, the response time of any task τ_i is bounded.

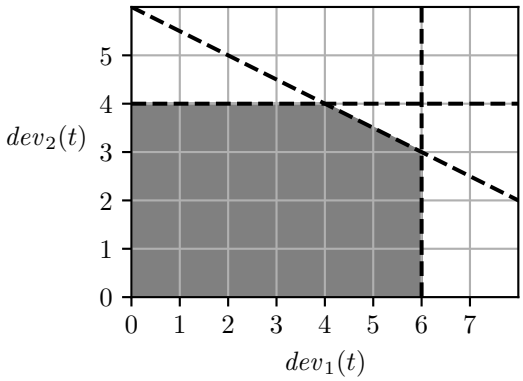
We describe our bounded region as the intersection of one or more inequalities. These inequalities have form $G(dev_1(t), dev_2(t), \ldots, dev_n(t)) \leq \beta$, where G is some $\mathbb{R}^n \mapsto \mathbb{R}$ function and β is a scalar. For example, suppose we have $\tau_{\text{act}}(t) = \{\tau_1, \tau_2\}$ over some time interval (we will discuss how changes in $\tau_{\text{act}}(t)$, *i.e.*, tasks entering or leaving the system, are dealt with later). The shaded area in Figure 3.1a illustrates the intersection of inequalities $dev_1(t) \leq 6$, $dev_2(t) \leq 4$, and $dev_1(t) + 2dev_2(t) \leq 12$. The dashed lines in Figure 3.1a illustrate the boundaries of these inequalities. In our proofs, these inequalities are specifically constructed such that we can draw conclusions about the configuration chosen by the scheduler at any time t where $\overrightarrow{\mathbf{dev}}(t)$ lies on one of these boundaries. These conclusions arise from the lemmas proven in Section 3.1.1.

We prove $\overrightarrow{\mathbf{dev}}(t)$ remains in our bounded region by contradiction. We assume otherwise that there exist time instants such that $\overrightarrow{\mathbf{dev}}(t)$ is outside of our bounded region. The proof by contradiction begins by showing that if there are time instants where $\overrightarrow{\mathbf{dev}}(t)$ is outside of the bounded region, then there exists a *boundary time instant*, denoted t_{b}, immediately prior to these time instants such that $\overrightarrow{\mathbf{dev}}(t_{\text{b}})$ lies on the boundary of the region.

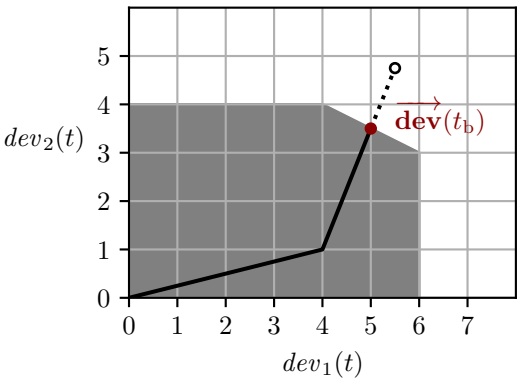
In Figure 3.1b, the trajectory of $\overrightarrow{\mathbf{dev}}(t)$ over time is illustrated as the black curve. The dotted segment illustrates a hypothetical continuation of this trajectory that reaches the white highlighted point at $(5.5, 4.75)$, which lies outside of our region. Prior to reaching this point, the trajectory intersects the boundary of the region at $(5, 3.5)$. The time instant corresponding with this point in the trajectory is t_{b}.

In our analysis, the existence of time t_{b} is proven using the limits of $dev_i(t)$ proven to exist in Corollaries 3.9 and 3.12 and the continuity properties of whichever function G corresponds with the portion of the region's boundary that $\overrightarrow{\mathbf{dev}}(t_{\text{b}})$ would lie on. Note that there may exist multiple such functions G if $\overrightarrow{\mathbf{dev}}(t_{\text{b}})$ lies on a corner of our bounded region (*e.g.*, $(4, 4)$ or $(6, 3)$ in Figure 3.1a), so our later proofs never assume that this function G is unique. In Figure 3.1b, the function G is unique and is $G(dev_1(t), dev_2(t)) = dev_1(t) + 2dev_2(t)$.

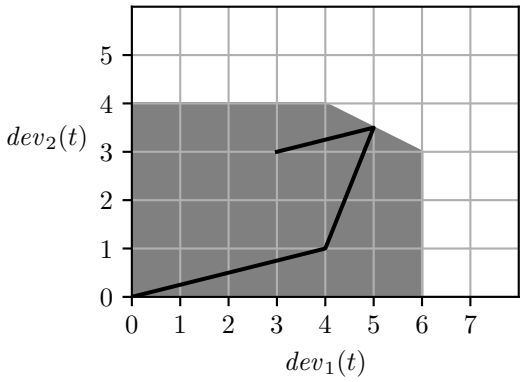
We next prove that the trajectory of $\overrightarrow{\mathbf{dev}}(t)$ remains within the region shortly after time t_{b}, which contradicts the definition of t_{b} (that t_{b} immediately precedes $\overrightarrow{\mathbf{dev}}(t)$ leaving the region). Recall from the earlier paragraphs in this subsection that the region is specifically constructed such that information is known about the configuration selected at time t when $\overrightarrow{\mathbf{dev}}(t)$ lies on the boundary of the region. We prove $\overrightarrow{\mathbf{dev}}(t)$



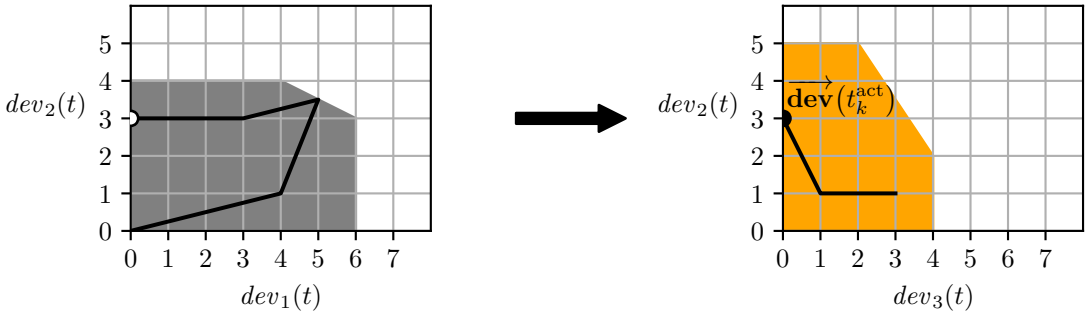
(a) Define a bounded region.



(b) Prove existence of t_b.



(c) Prove $\overrightarrow{\mathbf{dev}}(t)$ remains in region.



(d) Prove $\overrightarrow{\mathbf{dev}}(t)$ stays within region after task activation and deactivation.

Figure 3.1: Proof strategy.

remains within the region after t_b by using this information with the lemmas proven in Section 3.1.3, which concern how $dev_i(t)$ evolves over time. In Figure 3.1c, the trajectory $\overrightarrow{\mathbf{dev}}(t)$ moves to $(3, 3)$ after time t_b instead of to $(5.5, 4.75)$, the point highlighted in Figure 3.1b.

Contradicting the existence of t_b proves that $\overrightarrow{\mathbf{dev}}(t)$ remains in the bounded region, which is the goal of this proof strategy. As stated previously, the proof strategy assumes up to this point that the set of active tasks $\tau_{\text{act}}(t)$ remains constant. A dynamic $\tau_{\text{act}}(t)$ is dealt with by maintaining a different set of axes and a bounded region for each possible value of $\tau_{\text{act}}(t)$ considered (*e.g.*, for UNIFORM and IDENTICAL/ARBITRARY platforms, we will consider each subset $\tau' \subseteq \tau$ such that τ' is feasible on the considered platform).

For example, suppose that at some activation time instant t_k^{act}, task τ_1 from Figure 3.1 becomes inactive and another task τ_3 becomes active. The trajectory $\overrightarrow{\mathbf{dev}}(t)$ jumps from the left-side axes to the right-side axes in Figure 3.1d. Point $(0, 3)$ is highlighted white in the left-side vertical axis, and represents $\lim_{t \to (t_k^{\text{act}})^-} \overrightarrow{\mathbf{dev}}(t)$. Point $(0, 3)$ is highlighted black in the right-side vertical axis, and represents $\overrightarrow{\mathbf{dev}}(t_k^{\text{act}})$. Task set $\{\tau_2, \tau_3\}$ has a distinct bounded region, shaded orange in Figure 3.1d, from the bounded region of $\{\tau_1, \tau_2\}$, which is shaded gray in Figure 3.1.[2]

The obligation of the proof strategy is to show that $\overrightarrow{\mathbf{dev}}(t)$ always remains within the new bounded region after a jump. This is proven by induction on the activation time instants t_k^{act}. Once this is proven, we can derive response-time bounds that remain valid so long as $\tau_{\text{act}}(t)$ only takes whatever values we consider (*e.g.*, feasible subsets of tasks).

3.2 Analysis under $\mathcal{HP}$-$\mathcal{LAG}$ Systems

In this section, we will define a proposition on the considered task system, multiprocessor model, and scheduler, which we call $\mathcal{HP}$-$\mathcal{LAG}$. For systems where $\mathcal{HP}$-$\mathcal{LAG}$ is true, we will prove response-time bounds under WC schedulers. This will be useful because we will later prove that $\mathcal{HP}$-$\mathcal{LAG}$ is true for any feasible task system under UNIFORM or IDENTICAL/ARBITRARY. Thus, bounds proven in this section will apply to these multiprocessor models.

$\mathcal{HP}$-$\mathcal{LAG}$ depends on predicate $\mathcal{HP}$.

---

[2]Formally, these distinct axes and bounded regions all coexist within different hyperplanes in $\mathbb{R}^n$, but this is tedious to illustrate and does not seem to improve understanding. The switch from $\tau_{\text{act}}(t) = \{\tau_1, \tau_2\}$ to $\tau_{\text{act}}(t) = \{\tau_2, \tau_3\}$ is illustrated as a $\overrightarrow{\mathbf{dev}}(t)$ jumping between axes in Figure 3.1d, but in the space $\mathbb{R}^n$, the change in $\overrightarrow{\mathbf{dev}}(t)$ due to $\tau_{\text{act}}(t)$ is continuous.

▽ **Definition 3.3.** Predicate $\mathcal{HP}$ is

$$\mathcal{HP}\big(\tau', t\big) \triangleq \forall \tau_e \in \tau' :\ \forall \tau_\ell \in \tau_{\text{rdy}}(t) \setminus \tau' :\ pp_e(t) < pp_\ell(t). \qquad \triangle$$

In words, $\mathcal{HP}(\tau', t)$ states that at time t, all tasks in subset τ' have higher priority than any other ready task.

▽ **Definition 3.4.** Proposition $\mathcal{HP}$-$\mathcal{LAG}$ is

$$\mathcal{HP}\text{-}\mathcal{LAG} \triangleq \forall t :\ \forall \tau' \subseteq \tau_{\text{rdy}}(t) :\ \mathcal{HP}\big(\tau', t\big) \Rightarrow \sum_{\tau_i \in \tau'} csp_i(t) \geq U\big(\tau'\big). \qquad \triangle$$

In words, $\mathcal{HP}$-$\mathcal{LAG}$ states that at any time instant, for any subset of highest-priority ready tasks, the total speed of processors executing said tasks is at least the total utilization of said tasks.

▽ **Definition 3.5.** For each subset $\tau' \subseteq \tau$, let

$$\beta_{\tau'} \triangleq \frac{T_{[1]} + 2\phi}{2u_{[n]}} \left(U\big(\tau'\big)\right) \left(2U_{\text{max}} - U\big(\tau'\big)\right). \qquad \triangle$$

Static systems. Lemma 3.14, to be proven below, concerns a system where the set of active tasks is constant. Dynamic tasks will be considered afterwards.

▷ **Lemma 3.14.** Let $[t_0, t_1)$ be a time interval such that

$$\exists \tau^{\text{const}} \subseteq \tau :\ \forall t \in [t_0, t_1) :\ \tau_{\text{act}}(t) = \tau^{\text{const}}$$

and at time t_0, we have

$$\forall \tau' \subseteq \tau^{\text{const}} :\ \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i(t_0) \leq \beta_{\tau'}. \tag{3.13}$$

Under any $\mathcal{HP}$-$\mathcal{LAG}$ system, we have

$$\forall \tau' \subseteq \tau^{\text{const}} :\ \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i(t) \leq \beta_{\tau'}. \tag{3.14}$$

for any $t \in [t_0, t_1)$. ◁

Proof. We prove by contradiction. Suppose otherwise that there exist time instants in $[t_0, t_1)$ such that (3.14) is false. By (3.13), (3.14) is true at time t_0. Let $t_\mathrm{b} \in [t_0, t_1)$ denote the latest time instant such that (3.14) is true over $[t_0, t_\mathrm{b})$. We will show that the existence of t_b leads to a contradiction.

▶ **Claim 3.14.1.** $\forall \tau' \subseteq \tau^{\text{const}} :\ \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i(t_\mathrm{b}) \leq \beta_{\tau'}$. ◀

Proof. Consider any $\tau' \subseteq \tau^{\text{const}}$. By the definition of t_b, we have that

$$\forall t \in [t_0, t_b) : \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i(t) \leq \beta_{\tau'}.$$

Thus,

$$\begin{aligned}
\beta_{\tau'} &\geq \lim_{t^* \to t_\mathrm{b}^-} \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i(t^*) \\
&= \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot \lim_{t^* \to t_\mathrm{b}^-} dev_i(t^*) \\
&\geq \{\text{Lemma 3.10}\} \\
&\quad \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i(t_\mathrm{b}).
\end{aligned}$$ ■

▶ **Claim 3.14.2.** At time t_b, there exists $\tau^\mathrm{b} \subseteq \tau^{\text{const}}$ such that both

$$\forall \psi > 0 :\ \exists t^* \in (t_\mathrm{b}, t_\mathrm{b} + \psi) : \sum_{\tau_i \in \tau^\mathrm{b}} \sqrt{u_i} \cdot dev_i(t^*) > \beta_{\tau^\mathrm{b}} \text{ and} \tag{3.15}$$

$$\sum_{\tau_i \in \tau^\mathrm{b}} \sqrt{u_i} \cdot dev_i(t_\mathrm{b}) = \beta_{\tau^\mathrm{b}} \tag{3.16}$$

are true. ◀

Proof. We first prove (3.15) by contradiction. Suppose otherwise that

$$\forall \tau^\mathrm{b} \subseteq \tau^{\text{const}} :\ \exists \psi > 0 :\ \forall t \in (t_\mathrm{b}, t_\mathrm{b} + \psi) : \sum_{\tau_i \in \tau^\mathrm{b}} \sqrt{u_i} \cdot dev_i(t) \leq \beta_{\tau^\mathrm{b}}.$$

Because t_b is defined such that (3.14) is true over $[t_0, t_b)$ and $[t_0, t_b) \cup (t_b, t_b + \psi) = [t_0, t_b + \psi)$, we have

$$\forall \tau^b \subseteq \tau^{\text{const}} : \forall \in [t_0, t_b + \psi) : \sum_{\tau_i \in \tau^b} \sqrt{u_i} \cdot dev_i(t^*) \leq \beta_{\tau^b}.$$

This contradicts the definition of t_b as the latest time instant such that (3.14) is true over $[t_0, t_b)$.

It remains to prove (3.16). We have

$$\begin{aligned}
\sum_{\tau_i \in \tau^b} \sqrt{u_i} \cdot dev_i(t_b) = & \{\text{Corollary 3.12}\} \\
& \sum_{\tau_i \in \tau^b} \sqrt{u_i} \cdot \lim_{t^* \to t_b^+} dev_i(t^*) \\
= & \lim_{t^* \to t_b^+} \sum_{\tau_i \in \tau^b} \sqrt{u_i} \cdot dev_i(t^*) \\
\geq & \{\text{Equation (3.15)}\} \\
& \beta_{\tau^b}.
\end{aligned}$$

By Claim 3.14.1, we have

$$\sum_{\tau_i \in \tau^b} \sqrt{u_i} \cdot dev_i(t_b) \leq \beta_{\tau^b}.$$

Because $\sum_{\tau_i \in \tau^b} \sqrt{u_i} \cdot dev_i(t_b)$ is both upper and lower bounded by β_{τ^b}, (3.16) is the only possibility. This completes the proof of the claim. ■

► **Claim 3.14.3.** $\forall \tau_e \in \tau^b$:

$$\frac{dev_e(t_b)}{\sqrt{u_e}} \geq \frac{T_{[1]} + 2\phi}{2u_{[n]}} \left(2U_{\max} - 2U\left(\tau^b\right) + u_e\right) > 0. \quad \blacktriangleleft$$

Proof. We have

$$\begin{aligned}
& \sqrt{u_e} \cdot dev_e(t_b) \\
= & \sum_{\tau_i \in \tau^b} \sqrt{u_i} \cdot dev_i(t_b) - \sum_{\tau_i \in \tau^b \setminus \{\tau_e\}} \sqrt{u_i} \cdot dev_i(t_b)
\end{aligned}$$

$$
\begin{aligned}
&= \{\text{Claim 3.14.2, Equation (3.16)}\} \\
&\quad \beta_{\tau^{\mathrm{b}}} - \sum_{\tau_i \in \tau^{\mathrm{b}} \setminus \{\tau_e\}} \sqrt{u_i} \cdot dev_i(t_{\mathrm{b}}) \\
&\geq \{\text{Claim 3.14.1}\} \\
&\quad \beta_{\tau^{\mathrm{b}}} - \beta_{\tau^{\mathrm{b}} \setminus \{\tau_e\}} \\
&= \{\text{Definition 3.5}\} \\
&\quad \frac{T_{[1]} + 2\phi}{2u_{[n]}} \left(U\left(\tau^{\mathrm{b}}\right)\right) \left(2U_{\max} - U\left(\tau^{\mathrm{b}}\right)\right) \\
&\quad - \frac{T_{[1]} + 2\phi}{2u_{[n]}} \left(U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right)\right) \left(2U_{\max} - U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right)\right) \\
&= \left\{\text{By Definition 2.15, } U\left(\tau^{\mathrm{b}}\right) = U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right) + u_e\right\} \\
&\quad \frac{T_{[1]} + 2\phi}{2u_{[n]}} \left(U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right) + u_e\right) \left(2U_{\max} - U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right) - u_e\right) \\
&\quad - \frac{T_{[1]} + 2\phi}{2u_{[n]}} \left(U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right)\right) \left(2U_{\max} - U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right)\right) \\
&= \frac{T_{[1]} + 2\phi}{2u_{[n]}} \begin{pmatrix} 2U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right) \cdot U_{\max} - \left(U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right)\right)^2 - U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right) \cdot u_e \\ +2U_{\max} \cdot u_e - U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right) \cdot u_e - u_e^2 \end{pmatrix} \\
&\quad - \frac{T_{[1]} + 2\phi}{2u_{[n]}} \left(2U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right) \cdot U_{\max} - \left(U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right)\right)^2\right) \\
&= \frac{T_{[1]} + 2\phi}{2u_{[n]}} \left(-U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right) \cdot u_e + 2U_{\max} \cdot u_e - U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right) \cdot u_e - u_e^2\right) \\
&= \frac{T_{[1]} + 2\phi}{2u_{[n]}} \left(-U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right) + 2U_{\max} - U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right) - u_e\right) u_e \\
&= \frac{T_{[1]} + 2\phi}{2u_{[n]}} \left(2U_{\max} - 2U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right) - u_e\right) u_e \\
&= \left\{\text{By Definition 2.15, } -2U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right) = -2U\left(\tau^{\mathrm{b}}\right) + 2u_e\right\} \\
&\quad \frac{T_{[1]} + 2\phi}{2u_{[n]}} \left(2U_{\max} - 2U\left(\tau^{\mathrm{b}}\right) + 2u_e - u_e\right) u_e \\
&= \frac{T_{[1]} + 2\phi}{2u_{[n]}} \left(2U_{\max} - 2U\left(\tau^{\mathrm{b}}\right) + u_e\right) u_e \\
&> \left\{\text{By Definition 2.16, } U\left(\tau^{\mathrm{b}}\right) \leq U_{\max}\right\} \\
&\quad 0.
\end{aligned}
$$

Dividing by u_e yields the claim. ■

► **Claim 3.14.4.** $\forall \tau_\ell \in \tau^{\text{const}} \setminus \tau^{\text{b}}$:

$$\frac{dev_\ell(t_{\text{b}})}{\sqrt{u_\ell}} \leq \frac{T_{[1]}+2\phi}{2u_{[n]}} \left(2U_{\max} - 2U\left(\tau^{\text{b}}\right) - u_\ell\right). \qquad \blacktriangleleft$$

Proof. We have

$$\begin{aligned}
&\sqrt{u_\ell} \cdot dev_\ell(t_{\text{b}}) \\
=& \sum_{\tau_i \in \tau^{\text{b}} \cup \{\tau_\ell\}} \sqrt{u_i} \cdot dev_i(t_{\text{b}}) - \sum_{\tau_i \in \tau^{\text{b}}} \sqrt{u_i} \cdot dev_i(t_{\text{b}}) \\
=& \{\text{Claim 3.14.2, Equation (3.16)}\} \\
& \sum_{\tau_i \in \tau^{\text{b}} \cup \{\tau_\ell\}} \sqrt{u_i} \cdot dev_i(t_{\text{b}}) - \beta_{\tau^{\text{b}}} \\
\leq& \{\text{Claim 3.14.1}\} \\
& \beta_{\tau^{\text{b}} \cup \{\tau_\ell\}} - \beta_{\tau^{\text{b}}} \\
=& \{\text{Definition 3.5}\} \\
& \frac{T_{[1]}+2\phi}{2u_{[n]}} \left(U\left(\tau^{\text{b}} \cup \{\tau_\ell\}\right)\right) \left(2U_{\max} - U\left(\tau^{\text{b}} \cup \{\tau_\ell\}\right)\right) \\
& - \frac{T_{[1]}+2\phi}{2u_{[n]}} \left(U\left(\tau^{\text{b}}\right)\right) \left(2U_{\max} - U\left(\tau^{\text{b}}\right)\right) \\
=& \left\{\text{By Definition 2.15, } U\left(\tau^{\text{b}} \cup \{\tau_\ell\}\right) = U\left(\tau^{\text{b}}\right) + u_\ell\right\} \\
& \frac{T_{[1]}+2\phi}{2u_{[n]}} \left(U\left(\tau^{\text{b}}\right) + u_\ell\right) \left(2U_{\max} - U\left(\tau^{\text{b}}\right) - u_\ell\right) \\
& - \frac{T_{[1]}+2\phi}{2u_{[n]}} \left(U\left(\tau^{\text{b}}\right)\right) \left(2U_{\max} - U\left(\tau^{\text{b}}\right)\right) \\
=& \frac{T_{[1]}+2\phi}{2u_{[n]}} \begin{pmatrix} 2U\left(\tau^{\text{b}}\right) \cdot U_{\max} - \left(U\left(\tau^{\text{b}}\right)\right)^2 - U\left(\tau^{\text{b}}\right) \cdot u_\ell \\ +2U_{\max} \cdot u_\ell - U\left(\tau^{\text{b}}\right) \cdot u_\ell - u_\ell^2 \end{pmatrix} \\
& - \frac{T_{[1]}+2\phi}{2u_{[n]}} \left(2U\left(\tau^{\text{b}}\right) \cdot U_{\max} - \left(U\left(\tau^{\text{b}}\right)\right)^2\right) \\
=& \frac{T_{[1]}+2\phi}{2u_{[n]}} \left(-U\left(\tau^{\text{b}}\right) \cdot u_\ell + 2U_{\max} \cdot u_\ell - U\left(\tau^{\text{b}}\right) \cdot u_\ell - u_\ell^2\right) \\
=& \frac{T_{[1]}+2\phi}{2u_{[n]}} \left(-U\left(\tau^{\text{b}}\right) + 2U_{\max} - U\left(\tau^{\text{b}}\right) - u_\ell\right) u_\ell \\
=& \frac{T_{[1]}+2\phi}{2u_{[n]}} \left(2U_{\max} - 2U\left(\tau^{\text{b}}\right) - u_\ell\right) u_\ell.
\end{aligned}$$

Dividing by u_ℓ yields the claim. ■

► **Claim 3.14.5.** $\mathcal{HP}\big(\tau^{\mathrm{b}}, t_{\mathrm{b}}\big)$. ◄

Proof. Consider any task $\tau_e \in \tau^{\mathrm{b}}$ and $\tau_\ell \in \tau^{\text{const}} \setminus \tau^{\mathrm{b}}$. We have

$$
\begin{aligned}
&\frac{dev_e(t_{\mathrm{b}})}{\sqrt{u_e}} \\
\geq\ &\{\text{Claim 3.14.3}\} \\
&\frac{T_{[1]} + 2\phi}{2u_{[n]}} \left(2U_{\max} - 2U\big(\tau^{\mathrm{b}}\big) + u_e\right) \\
=\ &\frac{T_{[1]} + 2\phi}{2u_{[n]}} \left(2U_{\max} - 2U\big(\tau^{\mathrm{b}}\big)\right) + \frac{T_{[1]} + 2\phi}{2u_{[n]}} u_e \\
=\ &\frac{T_{[1]} + 2\phi}{2u_{[n]}} \left(2U_{\max} - 2U\big(\tau^{\mathrm{b}}\big) - u_\ell\right) + \frac{T_{[1]} + 2\phi}{2u_{[n]}} \left(u_e + u_\ell\right) \\
\geq\ &\{\text{Claim 3.14.4}\} \\
&\frac{dev_\ell(t_{\mathrm{b}})}{\sqrt{u_\ell}} + \frac{T_{[1]} + 2\phi}{2u_{[n]}} \left(u_e + u_\ell\right) \\
\geq\ &\left\{u_e + u_\ell \geq 2u_{[n]}\right\} \\
&\frac{dev_\ell(t_{\mathrm{b}})}{\sqrt{u_\ell}} + T_{[1]} + 2\phi.
\end{aligned}
$$

By Lemma 3.3, we have

$$\forall \tau_e \in \tau^{\mathrm{b}} :\ \forall \tau_\ell \in \tau^{\text{const}} \setminus \tau^{\mathrm{b}} :\ pp_e(t_{\mathrm{b}}) < pp_\ell(t_{\mathrm{b}}). \tag{3.17}$$

By Definitions 2.11 and 2.13, we have $\tau_{\text{rdy}}(t_{\mathrm{b}}) \subseteq \tau_{\text{act}}(t_{\mathrm{b}})$. By the lemma statement, and because $t_{\mathrm{b}} \in [t_0, t_1)$, we have $\tau_{\text{act}}(t_{\mathrm{b}}) = \tau^{\text{const}}$. Thus, we have $\tau_{\text{rdy}}(t_{\mathrm{b}}) \subseteq \tau^{\text{const}}$, which implies that for any $\tau_\ell \in \tau_{\text{rdy}}(t_{\mathrm{b}}) \setminus \tau^{\mathrm{b}}$, we have $\tau_\ell \in \tau^{\text{const}} \setminus \tau^{\mathrm{b}}$. The claim follows by (3.17) and Definition 3.3. ■

► **Claim 3.14.6.** $\sum_{\tau_i \in \tau^{\mathrm{b}}} csp_i(t_{\mathrm{b}}) \geq U\big(\tau^{\mathrm{b}}\big)$. ◄

Proof. By Claim 3.14.3, for any task $\tau_e \in \tau^{\mathrm{b}}$, we have $dev_e(t_{\mathrm{b}}) > 0$. By Lemma 3.1, for any task $\tau_e \in \tau^{\mathrm{b}}$, we have $\tau_e \in \tau_{\text{rdy}}(t_{\mathrm{b}})$.

The claim follows from Definition 3.4 and Claim 3.14.5. ■

By Lemma 3.13 and Claim 3.14.3, for any task $\tau_e \in \tau^{\mathrm{b}}$, there exists $\psi > 0$ such that $\forall t \in [t_{\mathrm{b}}, t_{\mathrm{b}}+\psi)$:

$$\sqrt{u_e} \cdot dev_e(t) \leq \sqrt{u_e} \cdot dev_e(t_{\mathrm{b}}) + (t - t_{\mathrm{b}}) \cdot (u_i - csp_e(t_{\mathrm{b}}))\,.$$

Summing over the tasks in τ^{b}, we have $\forall t \in [t_{\mathrm{b}}, t_{\mathrm{b}} + \psi)$:

$$\begin{aligned}
\sum_{\tau_e \in \tau^{\mathrm{b}}} \sqrt{u_e} \cdot dev_e(t) &\leq \sum_{\tau_e \in \tau^{\mathrm{b}}} \sqrt{u_e} \cdot dev_e(t_{\mathrm{b}}) + (t - t_{\mathrm{b}}) \cdot (u_i - csp_e(t_{\mathrm{b}})) \\
&= \left[\sum_{\tau_e \in \tau^{\mathrm{b}}} \sqrt{u_e} \cdot dev_e(t_{\mathrm{b}})\right] + (t - t_{\mathrm{b}}) \left[\sum_{\tau_e \in \tau^{\mathrm{b}}} (u_i - csp_e(t_{\mathrm{b}}))\right] \\
&= \{\text{Claim 3.14.2, Equation (3.16)}\} \\
&\quad \beta_{\tau^{\mathrm{b}}} + (t - t_{\mathrm{b}}) \left[\sum_{\tau_e \in \tau^{\mathrm{b}}} (u_i - csp_e(t_{\mathrm{b}}))\right] \\
&= \{\text{Definition 2.15}\} \\
&\quad \beta_{\tau^{\mathrm{b}}} + (t - t_{\mathrm{b}}) \left[U\left(\tau^{\mathrm{b}}\right) - \sum_{\tau_e \in \tau^{\mathrm{b}}} csp_e(t_{\mathrm{b}})\right] \\
&\leq \left\{t - t_{\mathrm{b}} \geq 0 \text{ and, by Claim 3.14.6, } U\left(\tau^{\mathrm{b}}\right) - \sum_{\tau_e \in \tau^{\mathrm{b}}} csp_e(t_{\mathrm{b}}) \leq 0\right\} \\
&\quad \beta_{\tau^{\mathrm{b}}}.
\end{aligned}$$

This contradicts (3.15) of Claim 3.14.2. This contradiction completes the proof of Lemma 3.14. □

Dynamic tasks. We will consider allowing tasks to enter and leave the system in Lemma 3.16.

Lemma 3.15 is used in the proof of Lemma 3.16 to compare the magnitudes of different $\beta_{\tau'}$.

▷ **Lemma 3.15.** Consider subsets τ^{sub} and τ^{sup} such that $\tau^{\mathrm{sub}} \subseteq \tau^{\mathrm{sup}} \subseteq \tau$. If $U(\tau^{\mathrm{sup}}) \leq U_{\max}$, then $U(\tau^{\mathrm{sup}}) \cdot (2U_{\max} - U(\tau^{\mathrm{sup}})) \geq U\left(\tau^{\mathrm{sub}}\right) \cdot \left(2U_{\max} - U\left(\tau^{\mathrm{sub}}\right)\right)$. ◁

Proof. Consider the function $f(x) \triangleq x \cdot (2U_{\max} - x)$. f has derivative $\frac{d}{dx} f = (2U_{\max} - x) + x \cdot (-1) = 2\,(U_{\max} - x)$. The derivative $\frac{d}{dx} f$ is non-negative while $x \leq U_{\max}$, *i.e.*, function f is non-decreasing with x over $(-\infty, U_{\max}]$. The lemma follows by substituting x with $U(\tau^{\mathrm{sup}})$ and $U\left(\tau^{\mathrm{sub}}\right)$. □

▷ **Lemma 3.16.** Under any $\mathcal{HP}$-$\mathcal{LAG}$ system, we have

$$\forall \tau' \subseteq \tau_{\text{act}}(t) : \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i(t) \leq \beta_{\tau'}$$

for any time t. ◁

Proof. We prove by induction on the activation time instants t_k^{act} for $k \in \mathbb{N}$. The induction hypothesis is as follows:

$$\forall t \in (-\infty, t_k^{\text{act}}] : \forall \tau' \subseteq \tau_{\text{act}}(t) : \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i(t) \leq \beta_{\tau'} \tag{3.18}$$

The base case of $k = 1$ is considered by the following claim.

► **Claim 3.16.1.** $\forall t \in (-\infty, t_1^{\text{act}}] : \forall \tau' \subseteq \tau_{\text{act}}(t) : \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i(t) \leq \beta_{\tau'}$. ◄

Proof. By Definition 2.14,

$$\forall t \in (-\infty, t_1^{\text{act}}) : \forall \tau' \subseteq \tau_{\text{act}}(t) : \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i(t) \leq \beta_{\tau'}$$

is vacuously true. It remains to prove that $\forall \tau' \subseteq \tau_{\text{act}}(t_1^{\text{act}}) : \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i(t_1^{\text{act}}) \leq \beta_{\tau'}$. Consider any $\tau' \subseteq \tau_{\text{act}}(t_1^{\text{act}})$. We have

$$\begin{aligned} \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i\big(t_1^{\text{act}}\big) \leq\ & \{\text{Lemma 3.10}\} \\ & \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot \lim_{t^* \to \left(t_1^{\text{act}}\right)^-} dev_i(t^*). \end{aligned} \tag{3.19}$$

By Definition 2.14, for any task τ_i and $t^* < t_1^{\text{act}}$, task τ_i is inactive at t^*. By Lemma 3.5, for any time $t^* < t_1^{\text{act}}$, $dev_i(t^*) = 0$. Thus, $\lim_{t^* \to \left(t_1^{\text{act}}\right)^-} dev_i(t^*) = 0$. Continuing from the derivation paused at (3.19), we have

$$\begin{aligned} \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i\big(t_1^{\text{act}}\big) &\leq \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot \lim_{t^* \to \left(t_1^{\text{act}}\right)^-} dev_i(t^*) \\ &= \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot 0 \end{aligned}$$

$$= 0$$

$$\leq \beta_{\tau'}.$$

This concludes the proof of Claim 3.16.1, the base case of the inductive proof of Lemma 3.16. ■

Our remaining obligation is to prove that (3.18) implies the $(k+1)^{\text{th}}$ case. This is split among the following two claims.

► **Claim 3.16.2.** (3.18) implies that

$$\forall t \in [t_k^{\text{act}}, t_{k+1}^{\text{act}}) : \forall \tau' \subseteq \tau_{\text{act}}(t) : \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i(t) \leq \beta_{\tau'}.$$ ◄

Proof. Let $\tau^{\text{const}} \triangleq \tau_{\text{act}}\big(t_k^{\text{act}}\big)$. By Definition 2.14, $\forall t \in [t_k^{\text{act}}, t_{k+1}^{\text{act}}) : \tau_{\text{act}}(t) = \tau^{\text{const}}$. The claim follows from (3.18) and Lemma 3.14 with $[t_0, t_1) = [t_k^{\text{act}}, t_{k+1}^{\text{act}})$. ■

► **Claim 3.16.3.** $\forall \tau' \subseteq \tau_{\text{act}}\big(t_{k+1}^{\text{act}}\big) : \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i\big(t_{k+1}^{\text{act}}\big) \leq \beta_{\tau'}.$ ◄

Proof. Consider any $\tau' \subseteq \tau_{\text{act}}\big(t_{k+1}^{\text{act}}\big)$. Let $\tau^{\text{old}} \triangleq \tau' \cap \tau_{\text{act}}\big(t_k^{\text{act}}\big)$ and $\tau^{\text{new}} \triangleq \tau' \setminus \tau_{\text{act}}\big(t_k^{\text{act}}\big)$. τ^{old} denotes tasks of τ' that were also active in $[t_k^{\text{act}}, t_{k+1}^{\text{act}})$, while τ^{new} denotes tasks of τ' that became active at time t_{k+1}^{act}.

We have

$$\begin{aligned}
&\sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i\big(t_{k+1}^{\text{act}}\big) \\
&= \left[\sum_{\tau_i \in \tau^{\text{old}}} \sqrt{u_i} \cdot dev_i\big(t_{k+1}^{\text{act}}\big)\right] + \left[\sum_{\tau_i \in \tau^{\text{new}}} \sqrt{u_i} \cdot dev_i\big(t_{k+1}^{\text{act}}\big)\right] \\
&\leq \{\text{Lemma 3.10}\} \\
&\quad \left[\sum_{\tau_i \in \tau^{\text{old}}} \sqrt{u_i} \cdot \lim_{t^* \to \left(t_{k+1}^{\text{act}}\right)^-} dev_i(t^*)\right] + \left[\sum_{\tau_i \in \tau^{\text{new}}} \sqrt{u_i} \cdot \lim_{t^* \to \left(t_{k+1}^{\text{act}}\right)^-} dev_i(t^*)\right] \\
&\leq \left\{\text{Claim 3.16.2 and } \tau^{\text{old}} \subseteq \tau_{\text{act}}\big(t_k^{\text{act}}\big)\right\} \\
&\quad \beta_{\tau^{\text{old}}} + \left[\sum_{\tau_i \in \tau^{\text{new}}} \sqrt{u_i} \cdot \lim_{t^* \to \left(t_{k+1}^{\text{act}}\right)^-} dev_i(t^*)\right]
\end{aligned}$$

$$= \{\text{Lemma 3.5 and } \tau_i \in \tau^{\text{new}} \Rightarrow \tau_i \text{ inactive over } [t_k^{\text{act}}, t_{k+1}^{\text{act}})\}$$

$$\beta_{\tau^{\text{old}}} + 0$$

$$\leq \{\text{By Definitions 2.16 and 3.5, Lemma 3.15, and } \tau^{\text{old}} \subseteq \tau' \subseteq \tau_{\text{act}}(t_{k+1}^{\text{act}})\}$$

$$\beta_{\tau'}.$$

This completes the proof of the claim. ■

Claims 3.16.2 and 3.16.3 form the induction step, thereby proving the induction hypothesis (3.18) for any $k \in \mathbb{N}$. Taking $k \to \infty$ yields the lemma statement. □

Theorem 3.17 presents our response-time bound for $\mathcal{HP}$-$\mathcal{LAG}$ systems.

▷ **Theorem 3.17.** For any $\mathcal{HP}$-$\mathcal{LAG}$ system, the response time of any task τ_i is at most

$$T_i + \frac{T_{[1]} + 2\phi}{2u_{[n]}} (2U_{\text{max}} - u_i).$$ ◁

Proof. Consider $dev_i(t)$ at any time instant t. There are two cases.

◀ **Case 3.17.1.** Task τ_i is inactive at t. ▶

By Lemma 3.5, $dev_i(t) = 0$. ♦

◀ **Case 3.17.2.** Task τ_i is active at t. ▶

By Definition 2.13, $\tau_i \in \tau_{\text{act}}(t)$. Thus, $\{\tau_i\} \subseteq \tau_{\text{act}}(t)$. By Lemma 3.16 and Definition 3.5, we have $\sqrt{u_i} \cdot dev_i(t) \leq \frac{T_{[1]}+2\phi}{2u_{[n]}} (2U_{\text{max}} - u_i) u_i$. ♦

In either case, we have $dev_i(t) \leq \frac{T_{[1]}+2\phi}{2u_{[n]}} (2U_{\text{max}} - u_i) \cdot \sqrt{u_i}$. The theorem follows from Lemma 3.4. □

3.3 Analysis under UNIFORM

We define Ufm-WC analogously to Ufm-EDF.

▷ **Ufm-WC.** At any time t, the ready task with earliest priority point is scheduled on the fastest processor, the ready task with second earliest priority point on the second fastest processor, and so on until all ready tasks are scheduled or all processors are scheduled upon. ◁

We prove response-time bounds under Ufm-WC by proving that Ufm-WC satisfies $\mathcal{HP}$-$\mathcal{LAG}$.

▷ **Lemma 3.18.** If, for task system τ executing under UNIFORM with a Ufm-WC scheduler, we have that for any time t, $\tau_{\text{act}}(t)$ is UNIFORM-Feasible, then this is an $\mathcal{HP}$-$\mathcal{LAG}$ system. ◁

Proof. For the duration of this proof, let $u_{[i](t)}$ denote the i^{th} largest utilization of any task in $\tau_{\text{act}}(t)$.

Consider any time t and subset $\tau' \subseteq \tau_{\text{rdy}}(t)$ such that $\mathcal{HP}(\tau', t)$. By Definition 3.4, it remains to prove that $\sum_{\tau_i \in \tau'} csp_i(t) \geq U(\tau')$.

By Definition 3.3, tasks of τ' have the earliest $pp_i(t)$ values at time t than any other task in $\tau_{\text{rdy}}(t)$. Under Ufm-WC, we have

$$\begin{aligned}
\sum_{\tau_i \in \tau'} csp_i(t) &= \sum_{j=1}^{\min\{m,|\tau'|\}} sp^{([j])} \\
&\geq \{\tau_{\text{act}}(t) \text{ is UNIFORM-Feasible at } t\} \\
&\quad \sum_{i=1}^{|\tau'|} u_{[i](t)} \\
&\geq \left\{\tau' \subseteq \tau_{\text{rdy}}(t) \subseteq \tau_{\text{act}}(t)\right\} \\
&\quad \sum_{\tau_i \in \tau'} u_i \\
&= \{\text{Definition 2.15}\} \\
&\quad U\big(\tau'\big).
\end{aligned}$$

By Definition 3.4, this is an $\mathcal{HP}$-$\mathcal{LAG}$ system. □

Corollary 3.19, which follows from Lemma 3.18 and Theorem 3.17, presents our response-time bound.

▷ **Corollary 3.19.** If, for task system τ executing under UNIFORM with a Ufm-WC scheduler, we have that at any time t, $\tau_{\text{act}}(t)$ is UNIFORM-Feasible, then the response time of any task τ_i is at most

$$T_i + \frac{T_{[1]} + 2\phi}{2u_{[n]}} \left(2U_{\max} - u_i\right). \quad \text{◁}$$

3.4 Analysis under IDENTICAL/ARBITRARY

We define Strong-APA-WC analogously to Strong-APA-EDF.

▷ **Strong-APA-WC.** At any time t, the chosen configuration is such that no unscheduled ready task τ_i has an alternating path beginning at τ_i and ending with either a processor that schedules no job or a job $\tau_{k,\ell}$ such that $pp_i(t) < pp_{k,j}(t)$.

Equivalently, at any time t, the chosen configuration corresponds with an optimal solution of $\text{MVM}\Big(\tau, \pi, \overrightarrow{\psi}, \mathbb{E}\Big)$ such that (2.11) and (2.12) are true and

$$\forall \tau_i, \tau_j \in \tau_{\text{rdy}}(t) : \; pp_i(t) < pp_j(t) \Rightarrow \psi_i > \psi_j, \tag{3.20}$$

i.e., tasks with earlier priority points have higher weight. ◁

The following lemmas will help to prove that Strong-APA-WC satisfies $\mathcal{HP}$-$\mathcal{LAG}$.

▷ **Lemma 3.20.** If $\tau_{\text{act}}(t)$ is IDENTICAL/ARBITRARY-Feasible, then for any $\tau' \subseteq \tau_{\text{act}}(t)$, optimization problem

$$\max \sum_{\tau_i \in \tau'} \sum_{\pi_j \in \alpha_i} y_{i,j} \text{ such that}$$

$$\forall \tau_i \in \tau' : \sum_{\pi_j \in \pi} y_{i,j} \leq 1.0 \tag{3.21}$$

$$\forall \pi_j \in \pi : \sum_{\tau_i \in \tau'} y_{i,j} \leq 1.0 \tag{3.22}$$

$$\mathbf{Y} \in \mathbb{R}_{\geq 0}^{n \cdot m} \tag{3.23}$$

has a solution such that

$$\sum_{\tau_i \in \tau'} \sum_{\pi_j \in \alpha_i} y_{i,j} \geq U\big(\tau'\big) \tag{3.24}$$

is true. ◁

Proof. By IDENTICAL/ARBITRARY-Feasible, $\exists \mathbf{X} \in \mathbb{R}^{n \cdot m}_{\geq 0}$ such that (2.9) and (2.10) are true. Let

$$y_{i,j} \triangleq \begin{cases} x_{i,j} \cdot u_i & \pi_j \in \alpha_i \\ 0 & \pi_j \notin \alpha_i \end{cases}.$$

By (2.9), we have

$$\forall \tau_i \in \tau_{\text{act}}(t) : \sum_{\pi_j \in \alpha_i} y_{i,j} = u_i. \tag{3.25}$$

Because $u_i \leq 1.0$ holds for each task $\tau_i \in \tau_{\text{act}}(t)$ and $\tau' \subseteq \tau_{\text{act}}(t)$, (3.25) implies (3.21). Because $\tau' \subseteq \tau_{\text{act}}(t)$, (2.10) (with $\tau \leftarrow \tau_{\text{act}}(t)$ because, as stated in the lemma, we are assuming that the subset $\tau_{\text{act}}(t)$ is IDENTICAL/ARBITRARY-Feasible) implies (3.22). Because $x_{i,j} \geq 0$ holds for each task $i \in \{1, 2, \ldots, n\}$ and $j \in \{1, 2, \ldots, m\}$, we also have (3.23). Summing (3.25) over the tasks in τ' yields $\sum_{\tau_i \in \tau'} \sum_{\pi_j \in \alpha_i} y_{i,j} = U(\tau')$, which satisfies (3.24). □

▷ **Lemma 3.21.** For an IDENTICAL/ARBITRARY-Feasible system, for any task subset $\tau' \subseteq \tau_{\text{act}}(t)$, there is a maximal matching $\mathbb{M}$ in the affinity graph between tasks in τ' and processors such that the size of the matching $|\mathbb{M}| \geq U(\tau')$. ◁

Proof. Consider the optimization problem in Lemma 3.20. This problem is an instance of AP. By Theorem 2.1, there is an optimal solution $\mathbf{Y}$ such that each $y_{i,j} \in \{0, 1\}$, *i.e.*, $\mathbf{Y}$ is binary. This binary $\mathbf{Y}$ represents a matching $\mathbb{M}$ in the affinity graph of the tasks in τ' and processors in π (recall Example 2.6). The number of matched tasks in $\tau_{\text{act}}(t)$ is $|\mathbb{M}| = \sum_{\tau_i \in \tau'} \sum_{\pi_j \in \alpha_i} y_{i,j}$, which is the objective function of the AP instance. By Lemma 3.20, this objective function has value at least $U(\tau')$, thus, $|\mathbb{M}| \geq U(\tau')$. □

The following lemma proves that Strong-APA-WC satisfies $\mathcal{HP}$-$\mathcal{LAG}$ if the system is always IDENTICAL/ARBITRARY-Feasible.

▷ **Lemma 3.22.** If, for task system τ executing under IDENTICAL/ARBITRARY with a Strong-APA-WC scheduler, we have that at any time t, $\tau_{\text{act}}(t)$ is IDENTICAL/ARBITRARY-Feasible, then this is an $\mathcal{HP}$-$\mathcal{LAG}$ system. ◁

Proof. Consider any time t and $\tau' \subseteq \tau_{\text{rdy}}(t)$ such that we have $\mathcal{HP}(\tau', t)$. It remains to show that $\sum_{\tau_i \in \tau'} csp_i(t) \geq U(\tau')$. Because, under IDENTICAL/ARBITRARY, each processor speed is 1.0, this

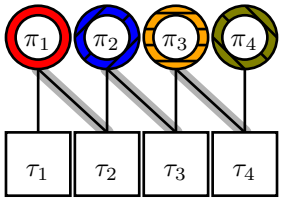


Figure 3.2: Example of an alternating path in a bipartite graph.

is equivalent to showing that the number of processors executing tasks of τ' is at least $U(\tau')$ (rounded up to the nearest whole number). Consider the matching $\mathbb{M}$ that corresponds with the configuration selected by Strong-APA-WC at time t. Because a task being matched is representative of said task being scheduled, the number of processors executing tasks of τ' is equal to the number of tasks in τ' that are matched in $\mathbb{M}$. Let $\mathbb{M}_{\tau'} \subseteq \mathbb{M}$ denote the subset of edges in $\mathbb{M}$ that are incident to tasks in τ. The cardinality $|\mathbb{M}_{\tau'}|$ is equal to the number of matched tasks in τ'. Thus, our proof obligation is to show that $|\mathbb{M}_{\tau'}| \geq U(\tau')$.

Consider the subgraph of the affinity graph made up of tasks of τ' and π (*i.e.*, τ', π, and the subset of edges in the affinity graph connecting τ' and π). The subset $\mathbb{M}_{\tau'}$ is a matching on this subgraph. To prove that $|\mathbb{M}_{\tau'}| \geq U(\tau')$, by Lemma 3.21, it is sufficient to show that matching $\mathbb{M}_{\tau'}$ is maximal for this subgraph.

We will show that $\mathbb{M}_{\tau'}$ is maximal for the subgraph of τ' by proving that $\mathbb{M}_{\tau'}$ contains no augmenting paths in the subgraph. Suppose otherwise that there is an augmenting path in $\mathbb{M}_{\tau'}$ on the subgraph of τ'. By Definition 2.25, this path is an alternating path originating from unmatched vertices (note that these vertices are specifically unmatched in the subgraph and may be matched in $\mathbb{M}_{\tau'}$ on the whole affinity graph). Because the subgraph is bipartite and this path is alternating, one of these unmatched vertices is a task in τ' and the other is a processor (see the example alternating path in Figure 3.2, which illustrates that every task and processor in the path besides endpoints τ_1 and π_4 must be matched to be in the alternating path). Because the subgraph only contains tasks of τ', the path begins with a task $\tau_e \in \tau'$. Let processor π_j denote the other endpoint of the path. Because π_j is unmatched in $\mathbb{M}_{\tau'}$ on the subgraph, in $\mathbb{M}_{\tau'}$ on the affinity graph, π_j is either also unmatched or matched to a task τ_ℓ in $\tau_{\text{rdy}}(t) \setminus \tau'$. Because we have assumed that τ' is such that $\mathcal{HP}(\tau', t)$ is true, we must have $pp_\ell(t) > pp_e(t)$. This

contradicts the definition of Strong-APA-WC. Thus, there are no augmenting paths in the subgraph of τ'. By Theorem 2.2, matching $\mathbb{M}_{\tau'}$ is maximal for the subgraph of τ'. □

Corollary 3.23, which follows from Lemma 3.22 and Theorem 3.17, presents our response-time bound.

▷ **Corollary 3.23.** If, for task system τ executing under IDENTICAL/ARBITRARY with a Strong-APA-WC scheduler, we have that at any time t, $\tau_{\text{act}}(t)$ is IDENTICAL/ARBITRARY-Feasible, then the response time of any task τ_i is at most

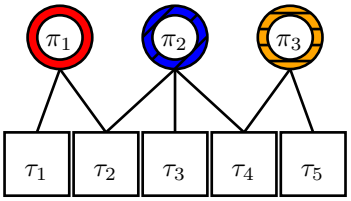
$$T_i + \frac{T_{[1]} + 2\phi}{2u_{[n]}} \left(2U_{\max} - u_i\right). \qquad \triangleleft$$

3.4.1 Counterexamples

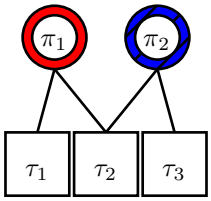
We briefly address the non-SRT-optimality of Weak-APA-EDF and non-preemptive Strong-APA-EDF under IDENTICAL/ARBITRARY, which we demonstrate by counterexample. Note that there is no need to address non-preemptivity under UNIFORM and UNRELATED because Ufm-EDF under UNIFORM (which is a special case of UNRELATED) with non-preemptive sections has already been shown to be non-SRT-optimal (Yang and Anderson, 2015). The non-SRT-optimality of Weak-APA-EDF is of interest because it is the author's belief that Weak-APA-EDF, due to its simpler requirements, is more likely to be implemented than Strong-APA-EDF in a real-time operating system (RTOS) with affinities.

▼ **Example 3.1.** Consider the task system whose affinity graph is illustrated in Figure 3.3a. Let $(C_1, T_1) = (C_5, T_5) = (2.0, 6.0)$, $(C_2, T_2) = (C_4, T_4) = (2.0, 2.0)$, and $(C_3, T_3) = (1.0, 6.0)$. This task system may have unbounded response times under a Weak-APA-EDF scheduler, as shown in Figure 3.4.

Tasks τ_2 and τ_4 release jobs periodically. Initially, tasks τ_2 and τ_4 execute on processors π_1 and π_3, respectively. At time 6.0, tasks τ_1 and τ_5 preempt tasks τ_2 and τ_4, respectively. The only other processor available to both tasks τ_2 and τ_4 is π_2, which they cannot both use. Both tasks have equal deadlines at time 6.0. We assume the tiebreak at time 6.0 favors task τ_2 and it is scheduled on processor π_2, while task τ_4 does not execute until time 8.0 when it resumes execution on processor π_3. Task τ_2 is also forced to migrate off of processor π_2 by task τ_3 at time 12.0. At time 18.0, tasks τ_1 and τ_5 again preempt tasks τ_2 and τ_4, respectively. Again, only π_2 is available to both tasks, except, unlike at time 6.0, τ_4 is scheduled



(a) Example 3.1 affinity graph.



(b) Example 3.2 affinity graph.

Figure 3.3: Counterexample affinity graphs.

over task τ_2 because it is tardy by 2.0 time units due to not being scheduled over $[6.0, 8.0)$. As a result, task τ_2 also becomes tardy by 2.0 time units by time 20.0. Thus, the deadlines of tasks τ_2 and τ_4 are once again equal.

The pattern of tasks τ_2 and τ_4 being simultaneously preempted by tasks τ_1 and τ_5 such that only one of τ_2 or τ_4 can be scheduled on processor π_2 can be repeated indefinitely, and with each occurrence, the maximum response time experienced by either task τ_2 or task τ_4 increases by 2.0 time units. ▲

▼ **Example 3.2.** Consider the system whose affinity graph is illustrated in Figure 3.3b. Let $(C_1, T_1) = (C_3, T_3) = (2.0, 4.0)$ and $(C_2, T_2) = (6.0, 6.0)$. A schedule over $[0, 120.0)$ for this system in which jobs are non-preemptive is illustrated in Figure 3.5.

Task τ_2 releases jobs periodically starting at time 0. Initially, task τ_2 executes on processor π_1. Task τ_1 releases its first job at time 1.0. Task τ_2 does not migrate to processor π_2 (which does not schedule a job at time 1.0) because it is non-preemptively executing its job. Task τ_3 releases its first job at time 5.0. Because π_2 is not executing any job, τ_3 is scheduled at time 5.0.

Once task τ_2 completes its job at time 6.0, its next job has a deadline of 12.0. This is later than the ready job of τ_1, which has a deadline of 5.0. Thus, task τ_1 is scheduled on processor π_1. Task τ_2 does not preempt task τ_3 on processor π_2 because it is executing non-preemptively. Task τ_2 only resumes executing at time 7.0 when task τ_3 completes its job. Because task τ_1 is not scheduled over $[6.0, 7.0)$, the response times of its jobs increases by 1.0 time unit.

Task τ_3 releases a job at time 9.0. Even though processor π_1 does not execute a job at time 9.0, task τ_2 does not migrate to π_1 because it is executing non-preemptively at time 9.0. When task τ_2 completes its job at time 13.0, it cannot execute on processor π_1 because task τ_1 is executing non-preemptively. Thus, task τ_1 is not scheduled over $[13.0, 14.0)$, increasing the response times of its jobs by 1.0 time unit.

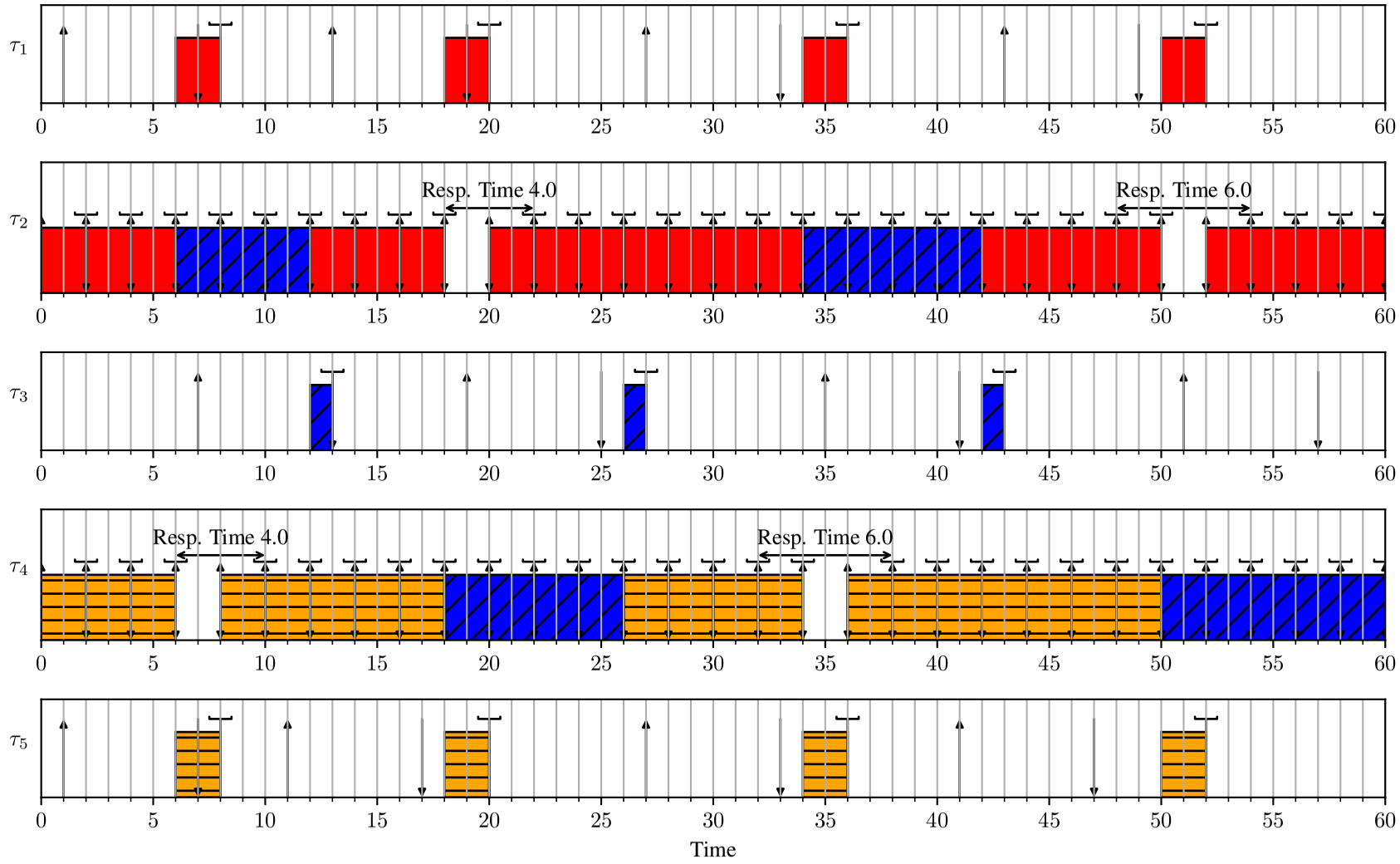


Figure 3.4: Weak-APA-EDF counterexample.

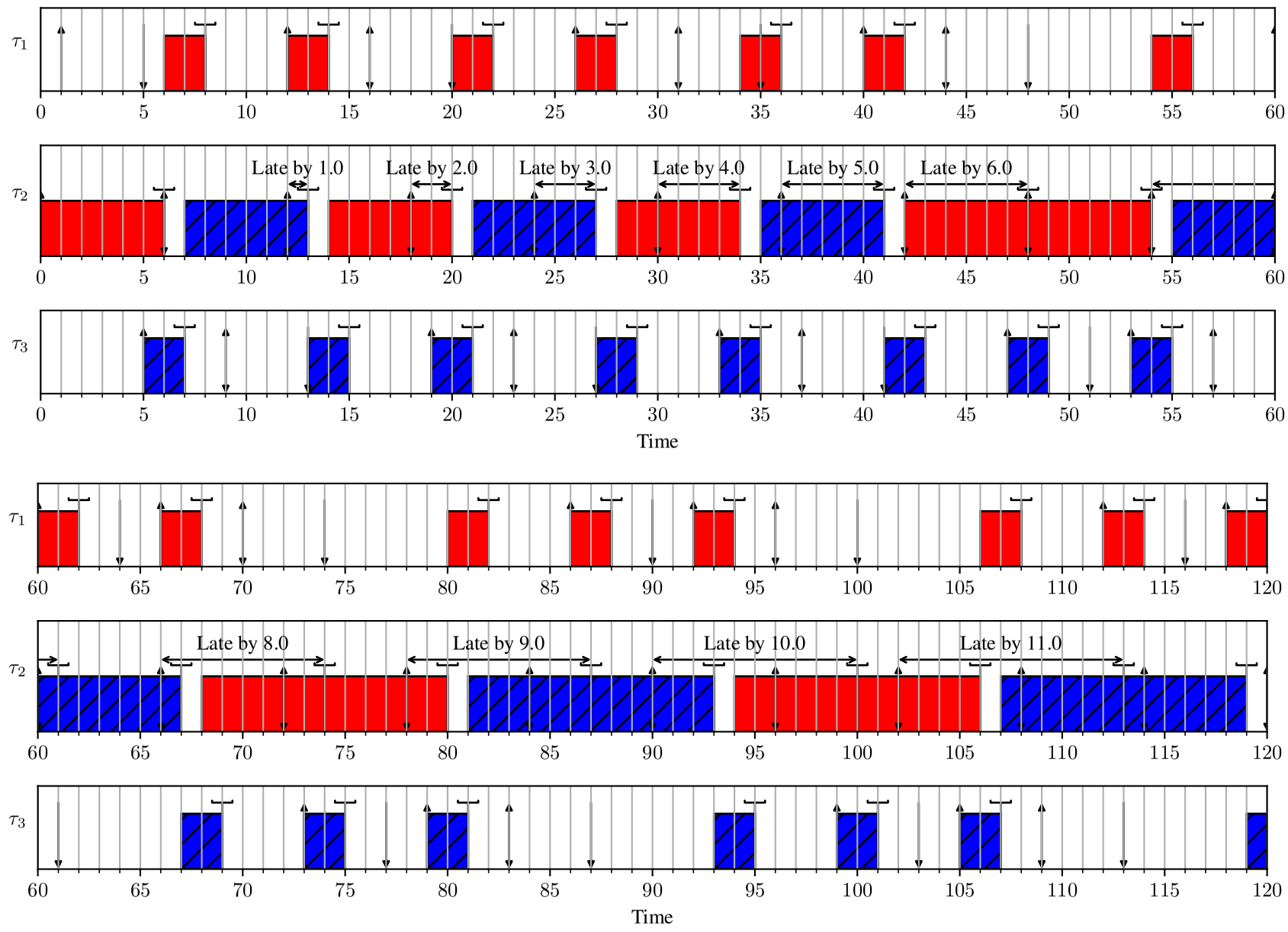


Figure 3.5: Non-preemptivity counterexample.

The schedule follows a pattern in which the response times of the jobs of task τ_2 increase by 1.0 time unit with every successive job. This is because whenever task τ_2 would otherwise migrate, the processor it would migrate to non-preemptively executes either task τ_1 or task τ_3, delaying the execution of τ_2 by 1.0 time unit.

This pattern changes at time 48.0. Task τ_2 completes a job at time 48.0 and continues executing its next job on processor π_1. This is because at time 48.0, both tasks τ_1 and τ_2 have deadlines of 48.0, and we assume the tie is broken in favor of task τ_2. Task τ_1 does not force task τ_2 to migrate until time 54.0, at which time the deadline of task τ_2 becomes $54.0 > 48.0$. Task τ_2 is unable to preempt task τ_3 executing on processor π_2, and is thus unscheduled over $[54.0, 55.0)$. The response times of jobs of task τ_2 are increased by 1.0 time unit.

In general, over $[0, 42.0)$, task τ_2 is prevented from executing for 1.0 time unit every job. This changes to every two jobs at time 42.0, and to every three jobs at some future time not shown in Figure 3.5. This is because task τ_2 is prevented from executing when it is forced to migrate by a job of task τ_1 or task τ_3 that is waiting for task τ_2 to vacate the corresponding processor (π_1 for task τ_1 and π_2 for task τ_3). For said job to force task τ_2 to migrate, it must wait until it has an earlier deadline than task τ_2. Task τ_2's deadline increases when it completes jobs. As task τ_2 becomes more tardy over time, the necessary number of completed jobs task τ_1 or task τ_3 must wait for increases.

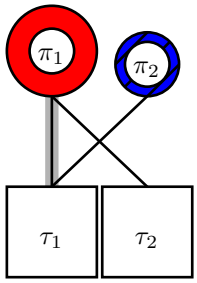
Though the time between durations where task τ_2 is delayed from executing increases over time, there are infinitely many such durations. Thus, the response time of task τ_2 is unbounded. ▲

3.5 Analysis under UNRELATED

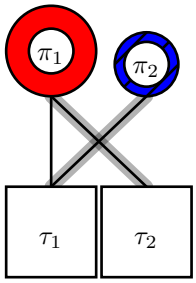
Limitations of $\mathcal{HP}$-$\mathcal{LAG}$. The strategy used to prove response-time bounds for Ufm-WC under UNIFORM and Strong-APA-WC under IDENTICAL/ARBITRARY for any feasible task system cannot be extended to schedulers under UNRELATED. This is because no scheduler satisfies $\mathcal{HP}$-$\mathcal{LAG}$ for any feasible task system under UNRELATED, as will be demonstrated by the following lemma.

▷ **Lemma 3.24.** Under UNRELATED, there exists no scheduler under which UNRELATED-Feasible implies the system satisfies $\mathcal{HP}$-$\mathcal{LAG}$. ◁

Proof. We prove the lemma by constructing a feasible task system under UNIFORM/ARBITRARY (a special case of UNRELATED) for which no scheduler satisfies $\mathcal{HP}$-$\mathcal{LAG}$. Consider tasks τ_1 and τ_2 with



(a) First configuration.



(b) Second configuration.

Figure 3.6: Both configurations violate $\mathcal{HP}$-$\mathcal{LAG}$ in Lemma 3.24.

$(C_1, T_1) = (3, 2)$ and $(C_2, T_2) = (4, 4)$. Then, $u_1 = 1.5$ and $u_2 = 1.0$. τ runs on two processors π_1 and π_2 with speeds $sp^{(1)} = 2.0$ and $sp^{(2)} = 1.0$. The tasks have the affinities illustrated in Figure 3.6.

Under the notation of UNRELATED, this multiprocessor has speeds

$$\begin{bmatrix} sp^{1,1} & sp^{1,2} \\ sp^{2,1} & sp^{2,2} \end{bmatrix} = \begin{bmatrix} sp^{(1)} & sp^{(2)} \\ sp^{(1)} & 0 \end{bmatrix} = \begin{bmatrix} 2.0 & 1.0 \\ 2.0 & 0 \end{bmatrix}.$$

We can see that this system is UNRELATED-Feasible because there exists

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{bmatrix} = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0 \end{bmatrix}$$

such that each of the following is true.

$$(2.14)\quad \begin{cases} sp^{1,1} \cdot x_{1,1} + sp^{1,2} \cdot x_{1,2} = & 2.0 \cdot 0.5 + 1.0 \cdot 0.5 \geq 1.5 & = u_1 \\ sp^{2,1} \cdot x_{2,1} + sp^{2,2} \cdot x_{2,2} = & 2.0 \cdot 0.5 + 1.0 \cdot 0 \geq 1.0 & = u_2 \end{cases}$$

$$(2.15)\quad \begin{cases} x_{1,1} + x_{1,2} = & 0.5 + 0.5 \leq & 1.0 \\ x_{2,1} + x_{2,2} = & 0.5 + 0 \leq & 1.0 \end{cases}$$

$$(2.16)\quad \begin{cases} x_{1,1} + x_{2,1} = & 0.5 + 0.5 \leq & 1.0 \\ x_{1,2} + x_{2,2} = & 0.5 + 0 \leq & 1.0 \end{cases}$$

Let time instant t be such that both tasks are ready with priority points $pp_1(t) < pp_2(t)$. By Definition 3.4, if a scheduler satisfies $\mathcal{HP}$-$\mathcal{LAG}$, then both

$$csp_1(t) \geq u_1, \text{ and} \tag{3.26}$$

$$csp_1(t) + csp_2(t) \geq u_1 + u_2 \tag{3.27}$$

are true.

However, any scheduler must choose one of the two configurations illustrated in Figure 3.6. This results in two cases.

◀ **Case 3.24.1.** The scheduler selects the configuration in Figure 2.4a. ▶

We have $csp_1(t) = 2.0$ and $csp_2(t) = 0$. Thus, $csp_1(t) + csp_2(t) = 2.0 + 0 < 1.5 + 1.0 = u_1 + u_2$, and (3.27) is violated. ♦

◀ **Case 3.24.2.** The scheduler selects the configuration in Figure 2.4b. ▶

We have $csp_1(t) = 1.0$ and $csp_2(t) = 2.0$. Thus, $csp_1(t) = 1.0 < 1.5 = u_1$, and (3.26) is violated. ♦

In either case, one of (3.26) or (3.27) is violated. Thus, this task system and multiprocessor, despite being UNRELATED-Feasible, do not satisfy $\mathcal{HP}$-$\mathcal{LAG}$ under *any* scheduler. □

3.5.1 Defining the Variant

This subsection defines a WC variant, Unr-WC, for UNRELATED. Our choice of definition for Unr-WC is justified by showing that both Ufm-WC and Strong-APA-WC are special cases of Unr-WC.

▽ **Definition 3.6.** The *profit* function of task $\tau_i \in \tau$ is

$$\Psi_i(t) \triangleq \begin{cases} t - pp_i(t) & t > pp_i(t) \text{ and } \tau_i \in \tau_{\text{rdy}}(t) \\ 0 & t \leq pp_i(t) \text{ or } \tau_i \notin \tau_{\text{rdy}}(t) \end{cases}. \qquad \triangle$$

▷ **Unr-WC.** At any time t, the configuration chosen is an optimal solution of AP$(\tau, \pi, \mathbf{P})$ in which

$$\mathbf{P} = \begin{bmatrix} \Psi_1(t) \cdot sp^{1,1} & \Psi_1(t) \cdot sp^{1,2} & \dots & \Psi_1(t) \cdot sp^{1,m} \\ \Psi_2(t) \cdot sp^{2,1} & \Psi_2(t) \cdot sp^{2,2} & & \\ \vdots & & \ddots & \\ \Psi_n(t) \cdot sp^{n,1} & & & \Psi_n(t) \cdot sp^{n,m} \end{bmatrix}. \qquad \triangleleft$$

Note that, for any time t, a canonical configuration $\mathbf{X}$ that is an optimal solution of the above AP instance always exists. Any optimal solution $\mathbf{X}$ can be transformed into a canonical configuration without modifying the objective function value. Suppose we have task τ_i and processor π_j such that $\tau_i \notin \tau_{\text{rdy}}(t)$ and $x_{i,j} = 1$. By Definition 3.6, $\Psi_i(t) = 0$. Thus, $p_{i,j} = \Psi_i(t) \cdot sp^{i,j} = 0 \cdot sp^{i,j} = 0$. Because $x_{i,j}$ is multiplied by $p_{i,j}$ in (2.1), the objective function value does not change by setting $x_{i,j} = 0$. Likewise, suppose we have task τ_i and processor π_j such that $\pi_j \notin \alpha_i$. Under UNRELATED, because τ_i does not have affinity for π_j, we have $sp^{i,j} = 0$. Thus, $p_{i,j} = \Psi_i(t) \cdot sp^{i,j} = \Psi_i(t) \cdot 0 = 0$. Again, setting $x_{i,j} = 0$ does not affect the objective function value. By Definition 2.26, $\mathbf{X}$ is canonical after setting such $x_{i,j}$ to 0.

After discussing Unr-WC in Section 3.5.1.1, we will show in Sections 3.5.1.2 and 3.5.1.3 that Ufm-WC and Strong-APA-WC are special cases of Unr-WC.

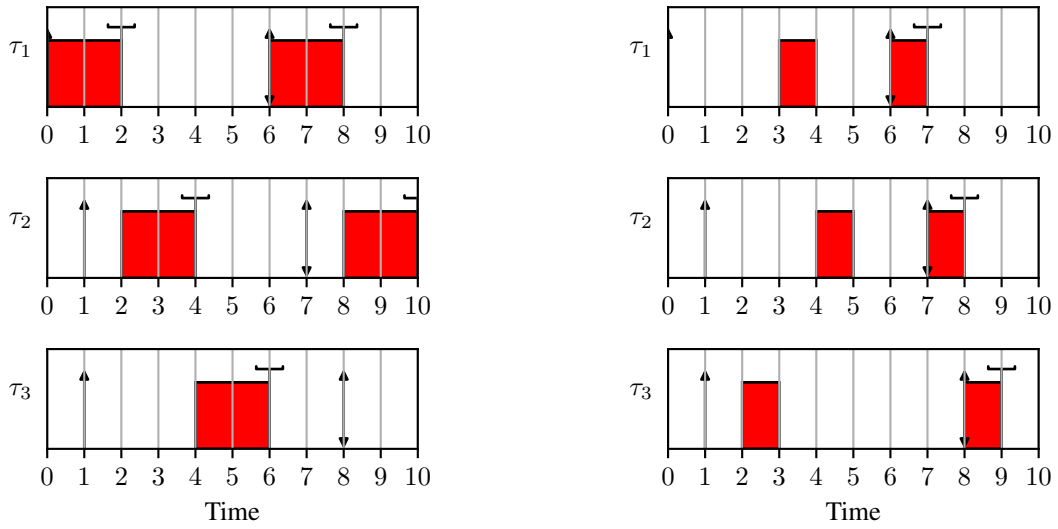
3.5.1.1 Interpreting Unr-WC

There are nuances to Unr-WC's behavior that are discussed in the following paragraphs.

Tasks with 0 profit. As seen in Definition 3.6, all tasks with priority points in the future have 0 profit when scheduled. Thus, a configuration that schedules such tasks is equally as profitable as a configuration that does not. Because Unr-WC only requires that the configuration chosen corresponds to some optimal solution of the AP instance for the current time instant, whether or not such tasks are scheduled is up to the specific implementation. This is illustrated in the following example.

▼ **Example 3.3.** Consider three tasks executing on a uniprocessor such that $(C_1, T_1) = (C_2, T_2) = (2.0, 6.0)$ and $(C_3, T_3) = (2.0, 7.0)$. Two schedules are illustrated for this system in Figure 3.7. Both schedules have the priority point of each task τ_i as $pp_i(t) = d_i(t)$.

The schedule in Figure 3.7a is a typical EDF schedule. Note that all illustrated jobs complete before their deadlines. Thus, for any task τ_i and time t in the illustrated time interval, we have $t - pp_i(t) =$



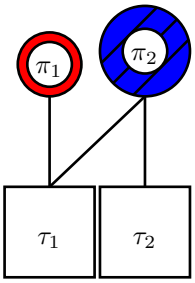
(a) EDF schedule.

(b) An Unr-WC schedule.

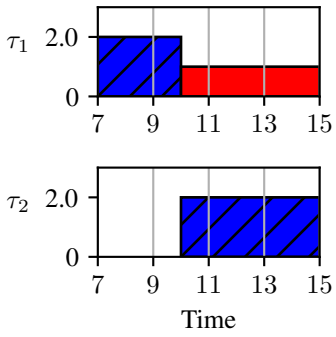
Figure 3.7: Scheduling of tasks with 0 profit.

$t - d_i(t) < 0$. Thus, scheduling any task at any time yields the same objective function value for Unr-WC's AP instance, meaning all configurations are equally optimal. Thus, scheduling the ready task with the earliest deadline at all times, as in EDF, is also optimal. This makes the EDF schedule a Unr-WC schedule over the pictured time interval.

The schedule in Figure 3.7b is also a Unr-WC schedule. Initially, as in the EDF schedule, over interval $[0, 6.0)$, all ready tasks have deadlines in the future, and thus yield 0 profit when scheduled. Thus, scheduling nothing over $[0, 2.0)$, task τ_3 over $[2.0, 3.0)$, task τ_1 over $[3.0, 4.0)$, and task τ_2 over $[4.0, 5.0)$ are all configurations corresponding to optimal solutions for the AP instances belonging to their corresponding time intervals. After time 6.0, when task τ_1 executes past its deadline, it is no longer the case that all tasks have 0 profit. For example, the profit of task τ_1 at time 6.5 is $\Psi_1(6.5) = 6.5 - d_i(6.5) = 6.5 - 6.0 = 0.5$. Scheduling task τ_1 at time 6.5 on the uniprocessor with speed 1.0 yields an objective function value of $1.0 \cdot 0.5 = 0.5$. Because scheduling this task now yields positive profit, the configuration that schedules task τ_1 on the only processor π_1 (*i.e.*, $\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{2,1} & x_{3,1} \end{bmatrix}^T = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^T$) is the only optimal solution to the AP instances over time interval $(6.0, 7.0)$. Thus, Unr-WC requires that task τ_1 is scheduled over $[6.0, 7.0)$ in Figure 3.7b. The same is true over $[7.0, 8.0)$ for task τ_2 and $[8.0, 9.0)$ for task τ_3. ▲



(a) Example 3.4 affinity graph.



(b) Example 3.4 schedule.

Figure 3.8: Example 3.4 illustration.

As shown in Example 3.3, configurations that do not schedule tasks may still be optimal under the AP instances defined by Unr-WC. This allows Unr-WC schedulers to not be work-conserving (*i.e.*, the scheduler may leave a ready task unscheduled, even if a processor is available). This does not prevent a specific Unr-WC scheduler from being work-conserving. For example, Ufm-WC and Strong-APA-WC (which we will show are special cases of Unr-WC) are both work-conserving.

Unpredictable migrations. Ufm-WC and Strong-APA-WC schedule only according to the relative order of tasks' priority points (*e.g.*, Ufm-WC schedules the ready task with earliest priority point on the fastest processor regardless of the magnitude of said priority point). Thus, Ufm-WC and Strong-APA-WC need only reschedule at time instants t when some task τ_i's priority point $pp_i(t)$ changes. This may not hold for Unr-WC under UNRELATED, as shown in the following example.

▼ **Example 3.4.** This example is illustrated by Figure 3.8. Consider a two-task and two-processor system with

$$\begin{bmatrix} sp^{1,1} & sp^{1,2} \\ sp^{2,1} & sp^{2,2} \end{bmatrix} = \begin{bmatrix} 1.0 & 2.0 \\ 0 & 2.0 \end{bmatrix}.$$

An affinity graph for this system is illustrated in Figure 3.8a. Suppose both tasks τ_1 and τ_2 are ready over $[7.0, 15.0)$ with $pp_1(t) = 0$ and $pp_2(t) = 5.0$. A schedule is presented in Figure 3.8b.

At time $t = 9.0$, we have $\Psi_1(t) = t - pp_1(t) = 9.0 - 0 = 9.0$ and $\Psi_2(t) = t - pp_2(t) = 9.0 - 5.0 = 4.0$. An optimal solution of the AP instance defined by Unr-WC at time 9.0 is $x_{1,2} = x_{2,1} = 1$ with objective value $\Psi_1(t) \cdot sp^{1,2} + \Psi_2(t) \cdot sp^{2,1} = 9.0 \cdot 2.0 + 4.0 \cdot 0 = 18.0$ (compared to $x_{1,1} = x_{2,2} = 1$

with value $\Psi_1(t) \cdot sp^{1,1} + \Psi_2(t) \cdot sp^{2,2} = 9.0 \cdot 1.0 + 4.0 \cdot 2.0 = 17.0$). Thus, at time 9.0, task τ_1 is scheduled on processor π_2 and task τ_2 is "scheduled" on processor π_1 (which it does not have affinity for).

However, at time $t = 11.0$, $\Psi_1(t) = t - pp_1(t) = 11.0 - 0 = 11.0$ and $\Psi_2(t) = t - pp_2(t) = 11.0 - 5.0 = 6.0$. The optimal solution at time 6.0 is then $x_{1,1} = x_{2,2} = 1$ with value $\Psi_1(t) \cdot sp^{1,1} + \Psi_2(t) \cdot sp^{2,2} = 11.0 \cdot 1.0 + 6.0 \cdot 2.0 = 23.0$ (compared to $x_{1,2} = x_{2,1} = 1$ with value $\Psi_1(t) \cdot sp^{1,2} + \Psi_2(t) \cdot sp^{2,1} = 11.0 \cdot 2.0 + 6.0 \cdot 0 = 22.0$). Thus, at time 11.0, task τ_1 is scheduled on processor π_1 and task τ_2 is scheduled on processor π_2.

Thus, a rescheduling occurs in $[7.0, 15.0)$ even though the tasks' priority points did not change. ▲

This makes Unr-WC impractical because rescheduling may occur at any time instant. The cause of this problem is that $\Psi_i(t)$ depends on t (see Definition 3.6), *i.e.*, $\Psi_i(t)$ varies continuously with time. Thus, the objective function value of a solution of the AP instance corresponding with Unr-WC varies continuously with time, which can cause a solution that is optimal at some time t_1 to be suboptimal at time t_2, even if $pp_i(t_1) = pp_i(t_2)$ for each task τ_i.

$\Psi_i(t)$ can be made to vary discretely with time using certain choices of priority point $pp_i(t)$. If, for each task τ_i, $\Psi_i(t)$ varies discretely with time, then the optimal solution for the AP instance corresponding with Unr-WC at a given time instant remains optimal until $\Psi_{i^*}(t)$ changes for some task τ_{i^*}. Thus, rescheduling only occurs at the time instants t when such a $\Psi_{i^*}(t)$ changes.

As an example of $\Psi_i(t)$ varying discretely with time for a certain choice of $pp_i(t)$, consider when $pp_i(t) = t - \left(\left\lfloor \frac{t}{T_i} \right\rfloor + 2\right) T_i + \tilde{d}_i(t)$. Task τ_i's profit would then be

$$
\begin{aligned}
\Psi_i(t) = & \ \{\text{Definition 3.6}\} \\
& \begin{cases} t - pp_i(t) & t > pp_i(t) \text{ and } \tau_i \in \tau_{\text{rdy}}(t) \\ 0 & t \leq pp_i(t) \text{ or } \tau_i \notin \tau_{\text{rdy}}(t) \end{cases} \\
= & \begin{cases} t - t + \left(\left\lfloor \frac{t}{T_i} \right\rfloor + 2\right) T_i - \tilde{d}_i(t) & t > t - \left(\left\lfloor \frac{t}{T_i} \right\rfloor + 2\right) T_i + \tilde{d}_i(t) \text{ and } \tau_i \in \tau_{\text{rdy}}(t) \\ 0 & t \leq t - \left(\left\lfloor \frac{t}{T_i} \right\rfloor + 2\right) T_i + \tilde{d}_i(t) \text{ or } \tau_i \notin \tau_{\text{rdy}}(t) \end{cases} \\
= & \begin{cases} \left(\left\lfloor \frac{t}{T_i} \right\rfloor + 2\right) T_i - \tilde{d}_i(t) & \left(\left\lfloor \frac{t}{T_i} \right\rfloor + 2\right) T_i > \tilde{d}_i(t) \text{ and } \tau_i \in \tau_{\text{rdy}}(t) \\ 0 & \left(\left\lfloor \frac{t}{T_i} \right\rfloor + 2\right) T_i \leq \tilde{d}_i(t) \text{ or } \tau_i \notin \tau_{\text{rdy}}(t) \end{cases},
\end{aligned}
$$

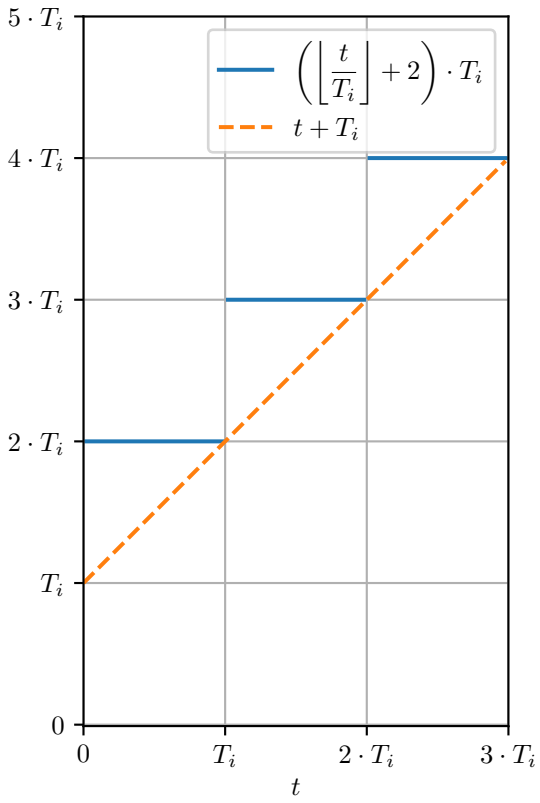


Figure 3.9: $\left(\left\lfloor \frac{t}{T_i} \right\rfloor + 2\right) T_i$ and $t + T_i$.

which, outside of changes in $\tilde{d}_i(t)$ (*i.e.*, job completions and arrivals), changes only every T_i time units (see the plot of $\left(\left\lfloor \frac{t}{T_i} \right\rfloor + 2\right) T_i$ in Figure 3.9). Because, in the worst case, jobs of task τ_i may arrive every T_i time units, the number of changes in $\Psi_i(t)$ over a time interval is asymptotically the same as the number of changes in $\tilde{d}_i(t)$ over the same interval. A timer that fires every T_i time units may be used to update $\Psi_i(t)$ and trigger rescheduling. Because $-2T_i \leq t - \left(\left\lfloor \frac{t}{T_i} \right\rfloor + 2\right) T_i < -T_i$, by Definition 2.20, this scheduler is WC with $\phi = 2T_{[1]}$.

Note that, on systems without early releasing, this choice of $pp_i(t)$ also mitigates that Unr-WC may be non-work-conserving. By Definition 2.11, without early releasing, a task τ_i being ready at time t (*i.e.*, $\tau_i \in \tau_{\text{rdy}}(t)$) implies that $t \geq a_i(t)$. By Definition 2.5, we have that $\tau_i \in \tau_{\text{rdy}}(t) \Rightarrow t + T_i \geq \tilde{d}_i(t)$. Because $\left(\left\lfloor \frac{t}{T_i} \right\rfloor + 2\right) T_i > t + T_i$ (see Figure 3.9), we have $\tau_i \in \tau_{\text{rdy}}(t) \Rightarrow \left(\left\lfloor \frac{t}{T_i} \right\rfloor + 2\right) T_i > \tilde{d}_i(t)$. Thus, $\tau_i \in \tau_{\text{rdy}}(t) \Rightarrow \Psi_i(t) > 0$. This prevents ready tasks from having 0 profit, which was the cause of the non-work-conserving behavior in Example 3.3.

3.5.1.2 Ufm-WC is a Special Case of Unr-WC

We now prove that Ufm-WC is a special case of Unr-WC. The following theorem will be useful for proving this.

▷ **Theorem 3.25 (Theorem 368 by Hardy et al. (1952)).** Let $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_n$ be two sequences. Then

$$\sum_{i=1}^{n} x_{[i]} \cdot y_{[n+1-i]} \leq \sum_{i=1}^{n} x_i \cdot y_i \leq \sum_{i=1}^{n} x_{[i]} \cdot y_{[i]},$$

i.e., the sum of products between elements of the sequences is greatest when the sequences are monotonically ordered in the same sense and least when monotonically ordered in the opposite sense. ◁

▷ **Lemma 3.26.** Ufm-WC on a UNIFORM multiprocessor is a special case of Unr-WC. ◁

Proof. We prove the lemma by showing that for any time t, any configuration selected at time t by Ufm-WC under UNIFORM also corresponds to a solution to the AP instance defined by Unr-WC. For the duration of this proof, we assume that $n = m$. This assumption can be made without loss of generality because under UNIFORM, if $n < m$, then only the n fastest processors need be considered, and if $n > m$, then we can analytically add $n - m$ processors with speed of 0.

► **Claim 3.26.1.** There exists an optimal solution $\mathbf{X}$ to the AP instance defined at time t by Unr-WC such that

$$\forall \tau_i \in \tau : \sum_{\pi_j \in \pi} x_{i,j} = 1.$$ ◄

Proof. Suppose otherwise that there exists task τ_{i^*} such that $\sum_{\pi_j \in \pi} x_{i^*,j} = 0$. Thus,

$$\begin{aligned}
\sum_{\pi_j \in \pi} \sum_{\tau_i \in \tau} x_{i,j} &= \sum_{\tau_i \in \tau} \sum_{\pi_j \in \pi} x_{i,j} \\
&= \sum_{\pi_j \in \pi} x_{i^*,j} + \sum_{\tau_i \in \tau \setminus \{\tau_{i^*}\}} \sum_{\pi_j \in \pi} x_{i,j} \\
&= 0 + \sum_{\tau_i \in \tau \setminus \{\tau_{i^*}\}} \sum_{\pi_j \in \pi} x_{i,j}
\end{aligned}$$

$$
\begin{aligned}
&\leq \{\text{Equation (2.2)}\} \\
&\quad \sum_{\tau_i \in \tau \setminus \{\tau_{i^*}\}} 1 \\
&= |\tau \setminus \{\tau_{i^*}\}| \\
&= n - 1.
\end{aligned}
$$

By (2.3), because $n = m$, and because $\sum_{\pi_j \in \pi} \sum_{\tau_i \in \tau} x_{i,j} \leq n - 1$, there exists π_{j^*} such that $\sum_{\tau_i \in \tau} x_{i,j^*} = 0$. Thus, x_{i^*,j^*} can be set to 1 without violating (2.2)-(2.4). Setting x_{i^*,j^*} to 1 changes the value of the objective function $\sum_{\tau_i \in \tau} \sum_{\pi_j \in \pi} \Psi_i(t) \cdot sp^{i,j} x_{i,j}$ by $\Psi_{i^*}(t) \cdot sp^{i^*,j^*}$. Because, by Definition 3.6, we have $\Psi_{i^*}(t) \geq 0$, the change to the objective function is non-negative.

This reasoning can be repeated for each task τ_i with $\sum_{\pi_j \in \pi} x_{i,j} = 0$, yielding the claim. ■

► **Claim 3.26.2.** The optimal solution $\mathbf{X}$ shown to exist in Claim 3.26.1 has objective function value

$$\sum_{i=1}^{n} \Psi_i(t) \cdot sp^{(j_i)}$$

for some distinct indices $j_1, j_2, \ldots, j_n \in \{1, 2, \ldots, n\}$. ◄

Proof. By Claim 3.26.1 and (2.4), for each $i \in \{1, 2, \ldots, n\}$, there exists unique $j \in \{1, 2, \ldots, n\}$ such that $x_{i,j} = 1$. Let $j_i \triangleq j$ when $x_{i,j} = 1$. The objective function value of $\mathbf{X}$ is then

$$
\begin{aligned}
&\sum_{\tau_i \in \tau} \sum_{\pi_j \in \pi} \Psi_i(t) \cdot sp^{i,j} \cdot x_{i,j} \\
&= \left\{\text{Under UNIFORM, } sp^{i,j} = sp^{(j)}\right\} \\
&\quad \sum_{\tau_i \in \tau} \sum_{\pi_j \in \pi} \Psi_i(t) \cdot sp^{(j)} \cdot x_{i,j} \\
&= \sum_{i=1}^{n} \sum_{\pi_j \in \pi} \Psi_i(t) \cdot sp^{(j)} \cdot x_{i,j} \\
&= \{x_{i,j} = 1 \Rightarrow j = j_i\} \\
&\quad \sum_{i=1}^{n} \Psi_i(t) \cdot sp^{(j_i)}.
\end{aligned}
$$

■

By Claim 3.26.2, any optimal solution to the AP instance defined by Unr-WC at time t has objective function value equal to the sum of element-wise products of sequences $\Psi_1(t), \Psi_2(t), \ldots, \Psi_n(t)$ and $sp^{(j_1)}, sp^{(j_2)}, \ldots, sp^{(j_n)}$. It remains to prove that the configuration chosen by Ufm-WC at time t yields an equivalent objective function value.

► **Claim 3.26.3.** The configuration selected by Ufm-WC at time t corresponds to a solution of AP with objective function value

$$\sum_{i=1}^{n} \Psi_{[i]}(t) \cdot sp^{([i])}. \qquad \blacktriangleleft$$

Proof. Let indices $i_1^*, i_2^*, \ldots, i_k^*, i_{k+1}^*, \ldots, i_n^*$ be such that

$$\tau_{\text{rdy}}(t) = \left\{ \tau_{i_1^*}, \tau_{i_2^*}, \ldots, \tau_{i_k^*} \right\} \tag{3.28}$$

and

$$pp_{i_1^*}(t) \leq pp_{i_2^*}(t) \leq \cdots \leq pp_{i_k^*}(t). \tag{3.29}$$

By Definition 3.6, (3.28), and (3.29), we have

$$\Psi_{i_1^*}(t) \geq \Psi_{i_2^*}(t) \geq \cdots \geq \Psi_{i_k^*}(t) \geq 0 = \Psi_{i_{k+1}^*}(t) = \Psi_{i_{k+2}^*}(t) = \cdots = \Psi_{i_n^*}(t). \tag{3.30}$$

Thus,

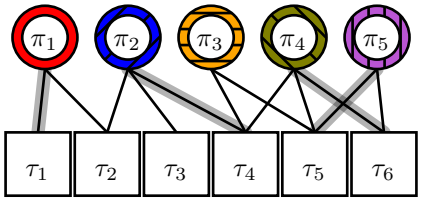
$$\forall r \in \{1, 2, \ldots, n\} : \ \Psi_{i_r^*}(t) = \Psi_{[r]}(t). \tag{3.31}$$

Consider the configuration chosen by Ufm-WC at time t. Task $\tau_{i_1^*}$ is scheduled on the processor with speed $sp^{([1])}$, task $\tau_{i_2^*}$ with speed $sp^{([2])}$, *etc.* Let $\mathbf{X}^*$ denote the solution of the AP instance such that $x_{i,j}^* = 1$ if task τ_i is scheduled on processor π_j under Ufm-WC. The objective function value of $\mathbf{X}^*$ is thus

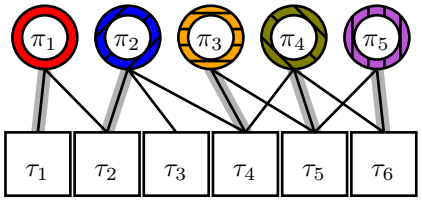
$$\sum_{\tau_i \in \tau} \sum_{\pi_j \in \pi} \Psi_i(t) \cdot sp^{i,j} \cdot x_{i,j}^*$$

$$
\begin{aligned}
&= \Big\{\text{Under UNIFORM, } sp^{i,j} = sp^{(j)}\Big\} \\
&\quad \sum_{\tau_i \in \tau} \sum_{\pi_j \in \pi} \Psi_i(t) \cdot sp^{(j)} \cdot x^*_{i,j} \\
&= \sum_{r=1}^{n} \sum_{\pi_j \in \pi} \Psi_{i^*_r}(t) \cdot sp^{(j)} \cdot x^*_{i^*_r,j} \\
&= \sum_{r=1}^{k} \sum_{\pi_j \in \pi} \Psi_{i^*_r}(t) \cdot sp^{(j)} \cdot x^*_{i^*_r,j} + \sum_{r=k+1}^{n} \sum_{\pi_j \in \pi} \Psi_{i^*_r}(t) \cdot sp^{(j)} \cdot x^*_{i^*_r,j} \\
&= \{\text{Ufm-WC}\} \\
&\quad \sum_{r=1}^{k} \Psi_{i^*_r}(t) \cdot sp^{([r])} + \sum_{r=k+1}^{n} \sum_{\pi_j \in \pi} \Psi_{i^*_r}(t) \cdot sp^{(j)} \cdot x^*_{i^*_r,j} \\
&= \{\text{Equation (3.31)}\} \\
&\quad \sum_{r=1}^{k} \Psi_{[r]}(t) \cdot sp^{([r])} + \sum_{r=k+1}^{n} \sum_{\pi_j \in \pi} \Psi_{i^*_r}(t) \cdot sp^{(j)} \cdot x^*_{i^*_r,j} \\
&= \{\text{Equation (3.30)}\} \\
&\quad \sum_{r=1}^{k} \Psi_{[r]}(t) \cdot sp^{([r])} + \sum_{r=k+1}^{n} 0 \\
&= \sum_{r=1}^{k} \Psi_{[r]}(t) \cdot sp^{([r])} + \sum_{r=k+1}^{n} 0 \cdot sp^{([r])} \\
&= \{\text{Equation (3.30)}\} \\
&\quad \sum_{r=1}^{k} \Psi_{[r]}(t) \cdot sp^{([r])} + \sum_{r=k+1}^{n} \Psi_{i^*_r}(t) \cdot sp^{([r])} \\
&= \{\text{Equation (3.31)}\} \\
&\quad \sum_{r=1}^{k} \Psi_{[r]}(t) \cdot sp^{([r])} + \sum_{r=k+1}^{n} \Psi_{[r]}(t) \cdot sp^{([r])} \\
&= \sum_{r=1}^{n} \Psi_{[r]}(t) \cdot sp^{([r])}. \qquad \blacksquare
\end{aligned}
$$

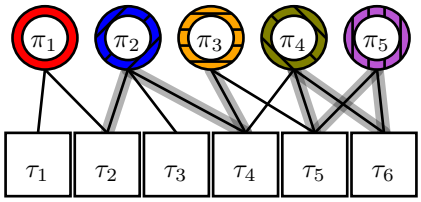
Using the above claims, we now complete the proof of the lemma. By Theorem 3.25, $\sum_{i=1}^{n} \Psi_{[i]}(t) \cdot sp^{([i])} \geq \sum_{i=1}^{n} \Psi_i(t) \cdot sp^{(j_i)}$. By Claims 3.26.2 and 3.26.3, the AP solution corresponding with the configuration chosen by Ufm-WC has objective function value at least that of any optimal solution. Thus,



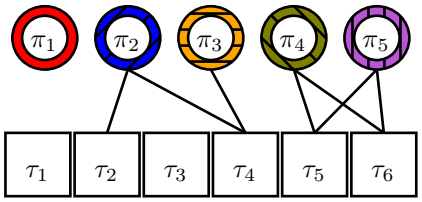
(a) Matching $\mathbb{M}$.



(b) Matching $\mathbb{M}'$.



(c) Symmetric difference $\mathbb{M}\Delta\mathbb{M}'$.



(d) Subgraph containing only edges in $\mathbb{M}\Delta\mathbb{M}'$.

Figure 3.10: Symmetric difference of matchings.

any configuration chosen by Ufm-WC at time t may also be chosen by Unr-WC, proving that Ufm-WC is a special case of Unr-WC. □

3.5.1.3 Strong-APA-WC is a Special Case of Unr-WC

We prove that Strong-APA-WC is a special case of Unr-WC in Lemma 3.29. The following definitions and lemmas will be used in that proof.

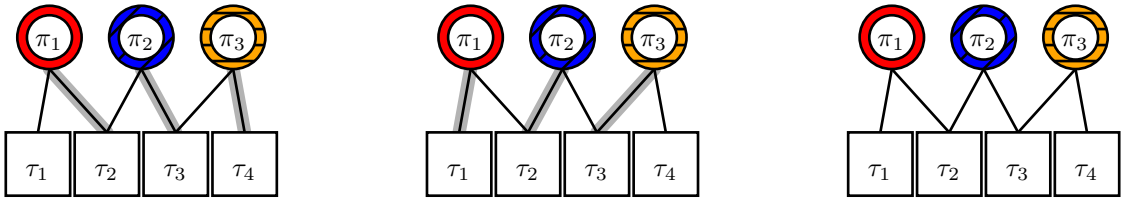
The symmetric difference and its properties discussed in Lemma 3.27 are well-known in graph theory.

▽ **Definition 3.7.** The *symmetric difference* between matchings $\mathbb{M}$ and $\mathbb{M}'$ is $\mathbb{M}\Delta\mathbb{M}' \triangleq (\mathbb{M} \cup \mathbb{M}') \setminus (\mathbb{M} \cap \mathbb{M}')$, *i.e.*, the set of edges in either matching that are not in both matchings. △

▼ **Example 3.5.** Figure 3.10 illustrates two matchings $\mathbb{M}$ (Figure 3.10a) and $\mathbb{M}'$ (Figure 3.10b) on the same graph, as well as the symmetric difference $\mathbb{M}\Delta\mathbb{M}'$ (Figure 3.10c). ▲

▷ **Lemma 3.27.** Consider matchings $\mathbb{M}$ and $\mathbb{M}'$ on a graph. Consider the subgraph that includes the edges of symmetric difference $\mathbb{M}\Delta\mathbb{M}'$ (Figure 3.10d). Every connected component of this subgraph is either

- an unconnected vertex (*e.g.*, τ_1 and π_1),
- a path (*e.g.*, $(\tau_2, \pi_2, \tau_4, \pi_3)$),



(a) Matching $\mathbb{M}$. (b) Matching $\mathbb{M}'$. (c) Edges of $\mathbb{M}\Delta\mathbb{M}'$.

Figure 3.11: Cases 3.28.2 and 3.28.3.

- or a cycle (*e.g.*, $(\tau_5, \pi_4, \tau_6, \pi_5, \tau_5)$). ◁

Proof. By Definition 2.22, any vertex is present in at most one edge in either matching. By Definition 3.7, every edge in the subgraph is present in either $\mathbb{M}$ or $\mathbb{M}'$. Thus, each vertex in the subgraph is incident with at most two edges. Given this, only the structures listed in the lemma statement are possible. □

▷ **Lemma 3.28.** Let matching $\mathbb{M}$ be an optimal solution for MVM$\left(\tau, \pi, \overrightarrow{\psi}, \mathbb{E}\right)$ with $\overrightarrow{\psi} \in \mathbb{R}^n_{\geq 0}$. Let $\overrightarrow{\psi}' \in \mathbb{R}^n_{\geq 0}$ be such that

$$\forall \tau_i \in \tau : \ \psi_i = 0 \Rightarrow \psi'_i = 0, \tag{3.32}$$

and

$$\forall \tau_i, \tau_j \in \tau : \ \psi_i \leq \psi_j \Rightarrow \psi'_i \leq \psi'_j, \tag{3.33}$$

i.e., the relative order of weights is the same in $\overrightarrow{\psi}$ and $\overrightarrow{\psi}'$. $\mathbb{M}$ is also an optimal solution for MVM$\left(\tau, \pi, \overrightarrow{\psi}', \mathbb{E}\right)$. ◁

Proof. We prove by contradiction. Suppose otherwise that $\mathbb{M}$ is an optimal solution for instance MVM$\left(\tau, \pi, \overrightarrow{\psi}, \mathbb{E}\right)$ and not for instance MVM$\left(\tau, \pi, \overrightarrow{\psi}', \mathbb{E}\right)$. Let τ^* denote the subset of matched tasks in $\mathbb{M}$. Let $\mathbb{M}'$ be an optimal solution of MVM$\left(\tau, \pi, \overrightarrow{\psi}', \mathbb{E}\right)$ such that any other optimal solution matches at most as many tasks of τ^* as $\mathbb{M}'$.

Consider the subgraph induced by $\mathbb{M}\Delta\mathbb{M}'$. Because $\mathbb{M}'$ is optimal for MVM$\left(\tau, \pi, \overrightarrow{\psi}', \mathbb{E}\right)$ and $\mathbb{M}$ is suboptimal, there exists some τ_i with $\psi'_i > 0$ such that τ_i is matched in $\mathbb{M}'$ and not in $\mathbb{M}$. By Lemma 3.27,

τ_i is the starting vertex of a path in the subgraph. There are three cases depending on how this path terminates.

◀ **Case 3.28.1.** The path in $\mathbb{M}\Delta\mathbb{M}'$ beginning with τ_i (*e.g.*, τ_2 in Figure 3.10d) terminates at a processor π_j (π_3 in Figure 3.10d). ▶

The path in $\mathbb{M}\Delta\mathbb{M}'$ beginning with τ_i is an augmenting path for matching $\mathbb{M}$. For instance $\text{MVM}\Big(\tau, \pi, \overrightarrow{\psi}, \mathbb{E}\Big)$, inverting the edges of this augmenting path (*e.g.*, in Figure 3.10d, adding (τ_2, π_2) and (τ_4, π_3) and removing (τ_4, π_2)) changes the weight of $\mathbb{M}_{\tau'}$ by ψ_i (ψ_2 in Figure 3.10d). By the contrapositive of (3.32) and because $\psi'_i > 0$, this change is positive. This contradicts that $\mathbb{M}$ is optimal for $\text{MVM}\Big(\tau, \pi, \overrightarrow{\psi}, \mathbb{E}\Big)$. ♦

Cases 3.28.2 and 3.28.3 are illustrated by matchings $\mathbb{M}$ and $\mathbb{M}'$ illustrated in Figures 3.11a and 3.11b, respectively, and the path $(\tau_1, \pi_1, \tau_2, \pi_2, \tau_3, \pi_3, \tau_4)$ in their symmetric difference $\mathbb{M}\Delta\mathbb{M}'$ (Figure 3.11c).

◀ **Case 3.28.2.** The path in $\mathbb{M}\Delta\mathbb{M}'$ beginning with τ_i (*e.g.*, τ_1 in Figure 3.11c) terminates at a task τ_k (τ_4 in Figure 3.11c) such that $\psi'_i > \psi'_k$. ▶

This path is an alternating path for matching $\mathbb{M}$ (*e.g.*, in Figure 3.11a, $(\tau_1, \pi_1) \notin \mathbb{M}$, $(\tau_2, \pi_1) \in \mathbb{M}$, $(\tau_2, \pi_2) \notin \mathbb{M}$, *etc.*). For $\text{MVM}\Big(\tau, \pi, \overrightarrow{\psi}, \mathbb{E}\Big)$, inverting the edges of this alternating path changes the weight of $\mathbb{M}_{\tau'}$ by $\psi_i - \psi_k$ (*e.g.*, inverting the edges in Figure 3.11a yields the matching in Figure 3.11b, which matches τ_1 and does not match τ_4, changing the total weight by $\psi_1 - \psi_4$). By the contrapositive of (3.33) and because $\psi'_i > \psi'_k$, this change is positive. This contradicts that $\mathbb{M}$ is optimal for $\text{MVM}\Big(\tau, \pi, \overrightarrow{\psi}, \mathbb{E}\Big)$. ♦

◀ **Case 3.28.3.** The path in $\mathbb{M}\Delta\mathbb{M}'$ beginning with τ_i terminates at a task τ_k such that $\psi'_i \leq \psi'_k$. ▶

This path is an alternating path for matching $\mathbb{M}'$ (*e.g.*, in Figure 3.11b, $(\tau_1, \pi_1) \in \mathbb{M}$, $(\tau_2, \pi_1) \notin \mathbb{M}$, $(\tau_2, \pi_2) \in \mathbb{M}$, *etc.*). Because this alternating path begins at τ_i (*e.g.*, τ_1 in Figure 3.11c), which is matched in $\mathbb{M}'$ and not matched in $\mathbb{M}$ (τ_1 is matched in Figure 3.11b and not in Figure 3.11a), and terminates at τ_k (τ_4), τ_k is matched in $\mathbb{M}$ and not matched in $\mathbb{M}'$ (τ_4 is matched in Figure 3.11a and not in Figure 3.11b). Thus, $\tau_k \in \tau^*$ (*e.g.*, $\tau_4 \in \{\tau_2, \tau_3, \tau_4\}$).

If $\psi'_i = \psi'_k$, then for $\text{MVM}\Big(\tau, \pi, \overrightarrow{\psi}', \mathbb{E}\Big)$, inverting the edges along the alternating path does not change the weight of $\mathbb{M}'$ and increases the number of tasks in τ^* that are matched in $\mathbb{M}'$ (*e.g.*,

inverting the edges in Figure 3.11b yields Figure 3.11a, in which task $\tau_4 \in \{\tau_2, \tau_3, \tau_4\}$ is matched). This contradicts that $\mathbb{M}'$ is the optimal matching that matches the most tasks in τ^*.

Otherwise, if $\psi'_i < \psi'_k$, then inverting the edges along the alternating path increases the weight of matched tasks by $\psi'_k - \psi_i$. This contradicts that $\mathbb{M}'$ is optimal for MVM$\Big(\tau, \pi, \overrightarrow{\psi}', \mathbb{E}\Big)$. ♦

All cases result in a contradiction. This proves the lemma. □

▷ **Lemma 3.29.** Strong-APA-WC on an IDENTICAL/ARBITRARY multiprocessor is a special case of Unr-WC. ◁

Proof. We prove the lemma by showing that at any time t, the configuration corresponding to the matching $\mathbb{M}$ that optimally solves the MVM instance defined by Strong-APA-WC also corresponds to an optimal solution to the AP instance defined by Unr-WC. Under IDENTICAL/ARBITRARY, this AP instance reduces to an MVM instance with weight $\psi_i = \Psi_i(t)$ for each task τ_i. At first glance, this is our proof obligation. This proof is incomplete because Strong-APA-WC requires that ψ_i satisfies (2.12) and (3.20), which both may be violated when $\psi_i = \Psi_i(t)$. (2.12) is violated when $\psi_i = \Psi_i(t)$ because a task $\tau_i \in \tau_{\text{rdy}}(t)$ with $t \leq pp_i(t)$ has $\Psi_i(t) = 0$ (by Definition 3.6). (3.20) is violated when $\psi_i = \Psi_i(t)$ because we may have tasks τ_i and τ_j in $\tau_{\text{rdy}}(t)$ such that $t \leq pp_i(t)$, $t \leq pp_j(t)$, and $pp_i(t) < pp_j(t)$, in which case we have $\Psi_i(t) = 0 = \Psi_j(t)$.

Our proof obligation is to show that an optimal matching under a $\overrightarrow{\psi}$ satisfying (2.12) and (3.20) remains optimal under $\Psi_i(t)$.

Consider any such $\overrightarrow{\psi}$. By (2.12) and Definition 3.6, we have

$$\forall \tau_i \in \tau : \ \psi_i = 0 \Rightarrow \Psi_i(t) = 0. \tag{3.34}$$

By the contrapositive of (3.20), we have $\forall \tau_i, \tau_j \in \tau_{\text{rdy}}(t) : \ \psi_i \leq \psi_j \Rightarrow pp_i(t) \geq pp_j(t)$. By Definition 3.6, we have

$$\forall \tau_i, \tau_j \in \tau_{\text{rdy}}(t) : \ \psi_i \leq \psi_j \Rightarrow \Psi_i(t) \leq \Psi_j(t). \tag{3.35}$$

By Definition 3.6, we have

$$\forall \tau_i \notin \tau_{\text{rdy}}(t) : \forall \tau_j \in \tau : \ \Psi_i(t) = 0 \leq \Psi_j(t). \tag{3.36}$$

By (2.12), we have

$$\forall \tau_i \in \tau_{\text{rdy}}(t) : \forall \tau_j \notin \tau_{\text{rdy}}(t) : \ \psi_i > \psi_j. \tag{3.37}$$

We next prove

$$\forall \tau_i, \tau_j \in \tau : \ \psi_i \leq \psi_j \Rightarrow \Psi_i(t) \leq \Psi_j(t) \tag{3.38}$$

using (3.35)-(3.37). Given tasks τ_i and τ_j, there are three cases for which of these tasks are in $\tau_{\text{rdy}}(t)$ at time t. If $\tau_i, \tau_j \in \tau_{\text{rdy}}(t)$, then (3.35) implies (3.38). If $\tau_i \notin \tau_{\text{rdy}}(t)$ and $\tau_j \in \tau_{\text{rdy}}(t)$ or $\tau_i, \tau_j \notin \tau_{\text{rdy}}(t)$, then by (3.36), we have $\Psi_i(t) \leq \Psi_j(t)$, which implies (3.38). If $\tau_i \in \tau_{\text{rdy}}(t)$ and $\tau_j \notin \tau_{\text{rdy}}(t)$, then by (3.37), we have $\psi_i > \psi_j$, which (as the negation of the premise $\psi_i \leq \psi_j$) implies (3.38). All cases imply (3.38).

The lemma follows from (3.34), (3.38), and Lemma 3.28. □

3.5.2 Response-Time Bounds

In this subsection, we derive response-time bounds under Unr-WC. These bounds asymptotically approach infinity as a parameter we denote the slowdown factor approaches 0.

▽ **Definition 3.8.** Task system τ has *slowdown factor* $s\ell \in (0, 1)$ if, for any time t, $\tau_{\text{act}}(t)$ is UNRELATED-Feasible when all processor speeds are multiplied by $(1.0 - s\ell)$, *i.e.*, at any time t, there exists $\mathbf{X} \in \mathbb{R}^{n \cdot m}_{\geq 0}$ such that

$$\forall \tau_i \in \tau_{\text{act}}(t) : \sum_{\pi_j \in \pi} (1.0 - s\ell) \cdot sp^{i,j} \cdot x_{i,j} \geq u_i \tag{3.39}$$

$$\forall \tau_i \in \tau_{\text{act}}(t) : \sum_{\pi_j \in \pi} x_{i,j} \leq 1.0 \tag{3.40}$$

$$\forall \pi_j \in \pi : \sum_{\tau_i \in \tau_{\text{act}}(t)} x_{i,j} \leq 1.0 \tag{3.41}$$

are true. △

Observe that $\tau_{\text{act}}(t)$ satisfying the UNRELATED-Feasible condition is equivalent to having a slowdown factor $s\ell = 0$. Having $s\ell > 0$ indicates that the system has excess capacity. Note that Definition 3.8 does not require that the same $\mathbf{X}$ satisfy (3.39)-(3.41) at every time instant. Time instants with distinct $\tau_{\text{act}}(t)$ are expected to have distinct $\mathbf{X}$'s.

The maximum speed of any processor in the system will also be a term in our derived bounds.

▽ **Definition 3.9.** The *maximum speed* is

$$sp^{\max} \triangleq \max_{\tau_i \in \tau, \pi_j \in \pi} \left\{ sp^{i,j} \right\}. \qquad \triangle$$

As in our analysis of $\mathcal{HP}$-$\mathcal{LAG}$ systems, our proof strategy for Unr-WC will be to upper bound some function of tasks' deviations. This upper bound will be β_{Unr}.

▽ **Definition 3.10.** For task system τ with slowdown factor $s\ell$, let

$$\beta_{\text{Unr}} \triangleq u_{[1]} \cdot \left(\frac{(n_{\text{act}}) \left(\phi + T_{[1]}\right) \left(sp^{\max} + u_{[n]}\right)}{s\ell \cdot u_{[n]}} \right)^2. \qquad \triangle$$

Static systems. As with our analysis of $\mathcal{HP}$-$\mathcal{LAG}$ systems, we first show in Lemma 3.34 that a function of tasks' deviations is bounded for an interval where the set of active tasks is constant before considering tasks that enter and leave dynamically. Lemmas 3.30-3.33 are used in the proof of Lemma 3.34.

▷ **Lemma 3.30.** For any time t, $\forall \tau_i \in \tau : \exists \delta > 0 : \forall t^* \in [t, t+\delta)$:

$$\begin{aligned} & \left(dev_i(t^*)\right)^2 \\ \leq\ & \left(dev_i(t)\right)^2 + 2(t^* - t) \cdot dev_i(t) \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}} \right) + (t^* - t)^2 \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}} \right)^2. \end{aligned} \qquad \triangleleft$$

Proof. Restrict δ to be small enough such that the current job of τ_i and $csp_i(t)$ are both constant over $[t, t+\delta)$ (as allowed by the Non-Fluid Assumption). There are three cases.

◀ **Case 3.30.1.** $t < vt_i(t)$. ▶

Further restrict δ such that $\delta \in (0, vt_i(t) - t)$. By Lemma 3.6, for any $t^* \in [t, t + \delta)$, we have $vt_i(t^*) - t^* \geq vt_i(t) - t^*$. Because $t^* - t < \delta < vt_i(t) - t$, we have $vt_i(t^*) - t^* > 0$. Thus, $t^* < vt_i(t^*)$.

By Definition 3.2 and because $t < vt_i(t)$ and $t^* < vt_i(t^*)$, we have $dev_i(t) = dev_i(t^*) = 0$. Thus,

$$
\begin{aligned}
& (dev_i(t^*))^2 \\
&= 0 \\
&= 0^2 + 2(t^* - t) \cdot 0 \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}}\right) \\
&= (dev_i(t))^2 + 2(t^* - t) \cdot dev_i(t) \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}}\right) \\
&\leq \{\text{Squares are non-negative}\} \\
& (dev_i(t))^2 + 2(t^* - t) \cdot dev_i(t) \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}}\right) + (t^* - t)^2 \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}}\right)^2.
\end{aligned}
$$

This is the lemma statement for this case. ♦

◄ **Case 3.30.2.** $t \geq vt_i(t)$ and $t^* \leq vt_i(t^*)$. ►

$$
\begin{aligned}
& (dev_i(t^*))^2 \\
&= \{\text{Definition 3.2}\} \\
& (\max\{0, \sqrt{u_i} \cdot (t^* - vt_i(t^*))\})^2 \\
&= \{t^* - vt_i(t^*) \leq 0\} \\
& 0 \\
&\leq \{\text{Squares are non-negative}\} \\
& \left(dev_i(t) + (t^* - t) \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}}\right)\right)^2 \\
&= (dev_i(t))^2 + 2(t^* - t) \cdot dev_i(t) \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}}\right) + (t^* - t)^2 \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}}\right)^2
\end{aligned}
$$

This is the lemma statement for this case. ♦

◀ **Case 3.30.3.** $t \geq vt_i(t)$ and $t^* > vt_i(t^*)$. ▶

$$
\begin{aligned}
& (dev_i(t^*))^2 \\
= \; & \{\text{Definition 3.2}\} \\
& (\max\{0, \sqrt{u_i} \cdot (t^* - vt_i(t^*))\})^2 \\
= \; & \{t^* - vt_i(t^*) \geq 0\} \\
& (\sqrt{u_i} \cdot (t^* - vt_i(t^*)))^2 \\
= \; & \{\text{Definition 3.1}\} \\
& \left(\sqrt{u_i} \cdot \left(t^* - a_i(t^*) - T_i \frac{c_i(t^*) - rem_i(t^*)}{c_i(t^*)}\right)\right)^2 \\
= \; & \{t^* \in [t, t+\delta) \text{ and the current job is constant over } [t, t+\delta)\} \\
& \left(\sqrt{u_i} \cdot \left(t^* - a_i(t) - T_i \frac{c_i(t) - rem_i(t^*)}{c_i(t)}\right)\right)^2 \\
= \; & \{t^* \in [t, t+\delta) \text{ and } csp_i(t) \text{ is constant over } [t, t+\delta)\} \\
& \left(\sqrt{u_i} \cdot \left(t^* - a_i(t) - T_i \frac{c_i(t) - rem_i(t) + (t^* - t) \cdot csp_i(t)}{c_i(t)}\right)\right)^2 \\
= \; & \left(\sqrt{u_i} \cdot \left(t - a_i(t) - T_i \frac{c_i(t) - rem_i(t)}{c_i(t)} + (t^* - t) - (t^* - t)\frac{T_i}{c_i(t)} csp_i(t)\right)\right)^2 \\
= \; & \{\text{Definition 3.1}\} \\
& \left(\sqrt{u_i} \cdot \left(t - vt_i(t) + (t^* - t) - (t^* - t)\frac{T_i}{c_i(t)} csp_i(t)\right)\right)^2 \\
= \; & \left(\sqrt{u_i} \cdot (t - vt_i(t)) + \sqrt{u_i} \cdot \left((t^* - t) - (t^* - t)\frac{T_i}{c_i(t)} csp_i(t)\right)\right)^2 \\
= \; & \{t \geq vt_i(t) \text{ and Definition 3.2}\} \\
& \left(dev_i(t) + \sqrt{u_i} \cdot \left((t^* - t) - (t^* - t)\frac{T_i}{c_i(t)} csp_i(t)\right)\right)^2 \\
\leq \; & \left\{-\frac{T_i}{c_i(t)} \leq -\frac{T_i}{C_i} = -\frac{1}{u_i}\right\} \\
& \left(dev_i(t) + \sqrt{u_i} \cdot \left((t^* - t) - (t^* - t)\frac{csp_i(t)}{u_i}\right)\right)^2 \\
= \; & (dev_i(t))^2 + 2(t^* - t) \cdot dev_i(t) \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}}\right) + (t^* - t)^2 \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}}\right)^2
\end{aligned}
$$

This is the lemma statement for this case. ♦

All cases yield the lemma statement. □

▷ **Lemma 3.31.** For any time t, if we have

$$\sum_{\tau_i \in \tau} dev_i(t) \cdot \left(\frac{csp_i(t)}{\sqrt{u_i}} - \sqrt{u_i} \right) > 0, \tag{3.42}$$

then $\exists \delta > 0 : \ \forall t^* \in [t, t+\delta) : \sum_{\tau_i \in \tau} \left(dev_i(t)\right)^2 \geq \sum_{\tau_i \in \tau} \left(dev_i(t^*)\right)^2$. ◁

Proof. By Lemma 3.30, for each task τ_i, there exists $\delta_i > 0$ such that $\forall t^* \in [t, t+\delta_i)$:

$$\begin{aligned} & \left(dev_i(t^*)\right)^2 \\ \leq \ & \left(dev_i(t)\right)^2 + 2(t^* - t) \cdot dev_i(t) \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}} \right) + (t^* - t)^2 \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}} \right)^2. \end{aligned} \tag{3.43}$$

Let $\delta \triangleq \min_{\tau_i \in \tau} \{\delta_i\}$. By the definition of δ and (3.43), we have $\forall t^* \in [t, t+\delta)$:

$$\left.\begin{aligned} & \sum_{\tau_i \in \tau} \left(dev_i(t^*)\right)^2 \\ \leq \ & \left[\sum_{\tau_i \in \tau} \left(dev_i(t)\right)^2 \right] + 2(t^* - t) \left[\sum_{\tau_i \in \tau} dev_i(t) \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}} \right) \right] \\ & + (t^* - t)^2 \left[\sum_{\tau_i \in \tau} \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}} \right)^2 \right]. \end{aligned} \right\} \tag{3.44}$$

We consider two cases depending on the value of $\sum_{\tau_i \in \tau} \left(\frac{csp_i(t)}{\sqrt{u_i}} - \sqrt{u_i} \right)^2$. Note this value, being a sum of squares, is non-negative.

◀ **Case 3.31.1.** $\sum_{\tau_i \in \tau} \left(\frac{csp_i(t)}{\sqrt{u_i}} - \sqrt{u_i} \right)^2 = 0.$ ▶

For any $t^* \in [t, t+\delta)$, we have

$$\begin{aligned} & \left(dev_i(t^*)\right)^2 \\ \leq \ & \{\text{Equation (3.44)}\} \\ & \left[\sum_{\tau_i \in \tau} \left(dev_i(t)\right)^2 \right] + 2(t^* - t) \left[\sum_{\tau_i \in \tau} dev_i(t) \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}} \right) \right] \end{aligned}$$

$$
\begin{aligned}
&+ (t^* - t)^2 \left[\sum_{\tau_i \in \tau} \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}} \right)^2 \right] \\
&= \left[\sum_{\tau_i \in \tau} \left(dev_i(t) \right)^2 \right] + 2(t^* - t) \left[\sum_{\tau_i \in \tau} dev_i(t) \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}} \right) \right] + (t^* - t)^2 \cdot 0 \\
&< \{\text{Equation (3.42)}\} \\
&\sum_{\tau_i \in \tau} \left(dev_i(t) \right)^2 .
\end{aligned}
$$

♦

◀ **Case 3.31.2.** $\sum_{\tau_i \in \tau} \left(\frac{csp_i(t)}{\sqrt{u_i}} - \sqrt{u_i} \right)^2 > 0.$ ▶

Let $\delta' \triangleq \min \left\{ \delta, \frac{2 \sum_{\tau_i \in \tau} dev_i(t) \cdot \left(\frac{csp_i(t)}{\sqrt{u_i}} - \sqrt{u_i} \right)}{\sum_{\tau_i \in \tau} \left(\frac{csp_i(t)}{\sqrt{u_i}} - \sqrt{u_i} \right)^2} \right\}$. Because $\delta > 0$ and (3.42), we have that $\delta' > 0$.
For any $t^* \in [t, t + \delta')$, we have

$$
\begin{aligned}
&\left(dev_i(t^*) \right)^2 \\
&\leq \left\{ \text{Equation (3.44) and } \delta' \in (0, \delta] \right\} \\
&\left[\sum_{\tau_i \in \tau} \left(dev_i(t) \right)^2 \right] \\
&+ 2(t^* - t) \left[\sum_{\tau_i \in \tau} dev_i(t) \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}} \right) \right] + (t^* - t)^2 \left[\sum_{\tau_i \in \tau} \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}} \right)^2 \right] \\
&= \left[\sum_{\tau_i \in \tau} \left(dev_i(t) \right)^2 \right] \\
&+ (t^* - t) \cdot \left(2 \left[\sum_{\tau_i \in \tau} dev_i(t) \cdot \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}} \right) \right] + (t^* - t) \left[\sum_{\tau_i \in \tau} \left(\sqrt{u_i} - \frac{csp_i(t)}{\sqrt{u_i}} \right)^2 \right] \right) \\
&\leq \left\{ t^* - t < \delta' \leq \frac{2 \sum_{\tau_i \in \tau} dev_i(t) \cdot \left(\frac{csp_i(t)}{\sqrt{u_i}} - \sqrt{u_i} \right)}{\sum_{\tau_i \in \tau} \left(\frac{csp_i(t)}{\sqrt{u_i}} - \sqrt{u_i} \right)^2} \right\} \\
&\left[\sum_{\tau_i \in \tau} \left(dev_i(t) \right)^2 \right] + (t^* - t) \cdot 0 \\
&= \sum_{\tau_i \in \tau} \left(dev_i(t) \right)^2 .
\end{aligned}
$$

♦

All cases yield the lemma statement. □

▷ **Lemma 3.32.** For any task τ_i and time t, we have $\left| \frac{dev_i(t)}{\sqrt{u_i}} - \Psi_i(t) \right| \leq T_{[1]} + \phi$. ◁

Proof. We have

$$\left.\begin{aligned} vt_i(t) - pp_i(t) &= \{\text{Definition 3.1}\} \\ &\quad a_i(t) + T_i \frac{c_i(t) - rem_i(t)}{c_i(t)} - pp_i(t) \\ &\leq \{rem_i(t) \geq 0\} \\ &\quad a_i(t) + T_i - pp_i(t) \\ &= \{\text{Definition 2.5}\} \\ &\quad \tilde{d}_i(t) - pp_i(t) \\ &\leq \{\text{Definition 2.20}\} \\ &\quad \phi \\ &\leq T_{[1]} + \phi, \end{aligned}\right\} \tag{3.45}$$

and

$$\left.\begin{aligned} pp_i(t) - vt_i(t) &= \{\text{Definition 3.1}\} \\ &\quad pp_i(t) - a_i(t) - T_i \frac{c_i(t) - rem_i(t)}{c_i(t)} \\ &\leq \{rem_i(t) \leq c_i(t)\} \\ &\quad pp_i(t) - a_i(t) \\ &\leq \{\text{Definition 2.5}\} \\ &\quad pp_i(t) - \tilde{d}_i(t) + T_i \\ &\leq \{\text{Definition 2.20}\} \\ &\quad \phi + T_i \\ &\leq \phi + T_{[1]}. \end{aligned}\right\} \tag{3.46}$$

There are four cases to consider.

◀ **Case 3.32.1.** $t > vt_i(t)$, $\tau_i \in \tau_{\text{rdy}}(t)$, and $t > pp_i(t)$. ▶

$$\left|\frac{dev_i(t)}{\sqrt{u_i}} - \Psi_i(t)\right| = \{\text{Definition 3.2}\}$$

$$|t - vt_i(t) - \Psi_i(t)|$$

$$
\begin{aligned}
&= \{\text{Definition 3.6}\} \\
&\quad |t - vt_i(t) - t + pp_i(t)| \\
&= |pp_i(t) - vt_i(t)| \\
&\leq \{\text{Equations (3.45) and (3.46)}\} \\
&\quad T_{[1]} + \phi
\end{aligned}
$$

♦

◄ **Case 3.32.2.** $t \leq vt_i(t)$, $\tau_i \in \tau_{\text{rdy}}(t)$, and $t > pp_i(t)$. ►

$$
\begin{aligned}
\left|\frac{dev_i(t)}{\sqrt{u_i}} - \Psi_i(t)\right| &= \{\text{Definition 3.2}\} \\
&\quad |0 - \Psi_i(t)| \\
&= \{\text{Definition 3.6}\} \\
&\quad t - pp_i(t) \\
&\leq \{\text{Definition 2.20}\} \\
&\quad t - \tilde{d}_i(t) + \phi \\
&= \{\text{Definition 2.5}\} \\
&\quad t - a_i(t) - T_i + \phi \\
&< \{rem_i(t) > 0\} \\
&\quad t - a_i(t) - T_i \frac{c_i(t) - rem_i(t)}{c_i(t)} + \phi \\
&= \{\text{Definition 3.1}\} \\
&\quad t - vt_i(t) + \phi \\
&\leq \{t \leq vt_i(t)\} \\
&\quad \phi \\
&\leq T_{[1]} + \phi
\end{aligned}
$$

♦

◄ **Case 3.32.3.** $t > vt_i(t)$ and either $\tau_i \notin \tau_{\text{rdy}}(t)$ or $t \leq pp_i(t)$. ►

Because $t > vt_i(t)$, by Definition 3.2 and Lemma 3.1, we have that $\tau_i \in \tau_{\text{rdy}}(t)$. Within this case, we must have $t \leq pp_i(t)$. Thus,

$$\begin{aligned}
\left|\frac{dev_i(t)}{\sqrt{u_i}} - \Psi_i(t)\right| &= \{\text{Definition 3.2}\} \\
&\quad |t - vt_i(t) - \Psi_i(t)| \\
&= \{\text{Definition 3.6}\} \\
&\quad |t - vt_i(t) - 0| \\
&= t - vt_i(t) \\
&\leq \{t \leq pp_i(t)\} \\
&\quad pp_i(t) - vt_i(t) \\
&\leq \{\text{Equation (3.46)}\} \\
&\quad T_{[1]} + \phi. \qquad \blacklozenge
\end{aligned}$$

◄ **Case 3.32.4.** $t \leq vt_i(t)$ and either $\tau_i \notin \tau_{\text{rdy}}(t)$ or $t \leq pp_i(t)$. ►

$$\begin{aligned}
\left|\frac{dev_i(t)}{\sqrt{u_i}} - \Psi_i(t)\right| &= \{\text{Definition 3.2}\} \\
&\quad |0 - \Psi_i(t)| \\
&= \{\text{Definition 3.6}\} \\
&\quad |0 - 0| \\
&= 0 \\
&\leq T_{[1]} + \phi \qquad \blacklozenge
\end{aligned}$$

The lemma follows in all cases. □

▷ **Lemma 3.33.** Consider $\tau_{\text{act}}(t)$ for some time t. The minimization problem (recall Definition 3.10)

$$\min \sum_{\tau_i \in \tau_{\text{act}}(t)} \frac{x_i}{\sqrt{u_i}} \text{ such that}$$

$$\sum_{\tau_i \in \tau_{\text{act}}(t)} x_i^2 = \beta_{\text{Unr}} \tag{3.47}$$

$$\overrightarrow{\mathbf{x}} \in \mathbb{R}^n_{\geq 0} \tag{3.48}$$

has optimal value at least

$$\sum_{\tau_i \in \tau_{\text{act}}(t)} \frac{x_i}{\sqrt{u_i}} \geq \frac{(n_{\text{act}})\left(\phi + T_{[1]}\right)\left(sp^{\max} + u_{[n]}\right)}{s\ell \cdot u_{[n]}}. \qquad \triangleleft$$

Proof. This problem is optimized when some $x_{i^*} = \sqrt{\beta_{\text{Unr}}}$ such that $u_{i^*} = \max_{\tau_k \in \tau_{\text{act}}(t)} \{u_k\}$, and $x_j = 0$ for all $j \neq i^*$.

We prove this by showing that the objective value of any other solution can be decreased.

▶ **Claim 3.33.1.** Let $\tau_{i^*}, \tau_j \in \tau_{\text{act}}(t)$ be two tasks such that for some solution vector $\overrightarrow{\mathbf{x}}$, we have $x_j > 0$. The vector $\overrightarrow{\mathbf{x}}'$ with k^{th} element

$$x'_k \triangleq \begin{cases} 0 & k = j \\ \sqrt{x_{i^*}^2 + x_j^2} & k = i^* \\ x_k & k \neq i^* \text{ and } k \neq j \end{cases} \tag{3.49}$$

is also a solution. ◀

Proof. We need to show that $\overrightarrow{\mathbf{x}}'$ satisfies (3.47) and (3.48). (3.48) is true because each case for the value of x'_k is non-negative, *i.e.*, $0 \geq 0$, $\sqrt{x_{i^*}^2 + x_j^2} \geq 0$, and $x_k \geq 0$ (because (3.48) was true for the original solution $\overrightarrow{\mathbf{x}}$).

For (3.47), we have

$$\begin{aligned} \sum_{\tau_k \in \tau_{\text{act}}(t)} \left(x'_k\right)^2 &= \left(x'_{i^*}\right)^2 + \left(x'_j\right)^2 + \sum_{\tau_k \in \tau_{\text{act}}(t) \setminus \{\tau_{i^*}, \tau_j\}} x_k^2 \\ &= \{\text{Equation (3.47)}\} \\ &\quad \left(x'_{i^*}\right)^2 + \left(x'_j\right)^2 + \beta_{\text{Unr}} - x_{i^*}^2 - x_j^2 \\ &= \{\text{Equation (3.49)}\} \\ &\quad \left(x_{i^*}^2 + x_j^2\right) + 0^2 + \beta_{\text{Unr}} - x_{i^*}^2 - x_j^2 \end{aligned}$$

$$= \beta_{\text{Unr}}. \qquad \blacksquare$$

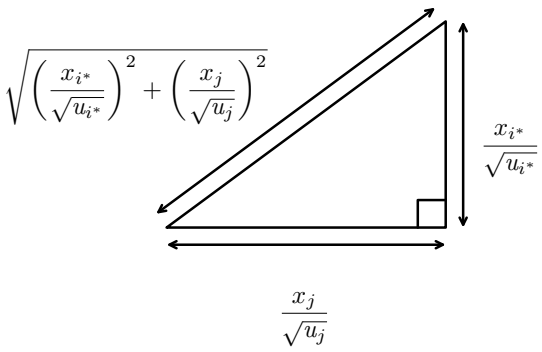
► **Claim 3.33.2.** Let $\tau_{i^*}, \tau_j \in \tau_{\text{act}}(t)$ be two tasks such that for some solution $\overrightarrow{\mathbf{x}}$, we have $x_j > 0$ and $u_{i^*} \geq u_j$. Solution $\overrightarrow{\mathbf{x}}'$ as defined in Claim 3.33.1 has an equal or lower objective value than $\overrightarrow{\mathbf{x}}$. ◄

Proof. Consider $\frac{x_{i^*}}{\sqrt{u_{i^*}}}$ and $\frac{x_j}{\sqrt{u_j}}$ to be the length of the legs of a (possibly degenerate if $x_{i^*} = 0$) right triangle (see Figure 3.12a). By the Pythagorean Theorem, the length of the hypotenuse is $\sqrt{\left(\frac{x_{i^*}}{\sqrt{u_{i^*}}}\right)^2 + \left(\frac{x_j}{\sqrt{u_j}}\right)^2}$. By the Triangle Inequality, the length of any side of a triangle is at most the sum of the lengths of the other two sides (see Figure 3.12b). Then

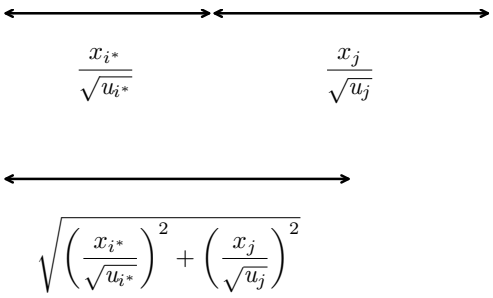
$$\left.\begin{aligned} \frac{x_{i^*}}{\sqrt{u_{i^*}}} + \frac{x_j}{\sqrt{u_j}} &\geq \{\text{Pythagorean Theorem and Triangle Inequality}\} \\ &\quad \sqrt{\left(\frac{x_{i^*}}{\sqrt{u_{i^*}}}\right)^2 + \left(\frac{x_j}{\sqrt{u_j}}\right)^2} \\ &= \frac{1}{\sqrt{u_{i^*}}}\sqrt{x_{i^*}^2 + \frac{u_{i^*}}{u_j}x_j^2} \\ &\geq \{u_{i^*} \geq u_j \text{ and } x_j > 0\} \\ &\quad \frac{1}{\sqrt{u_{i^*}}}\sqrt{x_{i^*}^2 + x_j^2}. \end{aligned}\right\} \tag{3.50}$$

Thus, the objective value of $\overrightarrow{\mathbf{x}}'$ is

$$\begin{aligned} \sum_{\tau_k \in \tau_{\text{act}}(t)} \frac{x'_k}{\sqrt{u_k}} &= \frac{x'_{i^*}}{\sqrt{u_{i^*}}} + \frac{x'_j}{\sqrt{u_j}} + \sum_{\tau_k \in \tau_{\text{act}}(t) \setminus \{\tau_{i^*}, \tau_j\}} \frac{x'_k}{\sqrt{u_k}} \\ &= \{\text{Equation (3.49)}\} \\ &\quad \frac{\sqrt{x_{i^*}^2 + x_j^2}}{\sqrt{u_{i^*}}} + \frac{0}{\sqrt{u_j}} + \sum_{\tau_k \in \tau_{\text{act}}(t) \setminus \{\tau_{i^*}, \tau_j\}} \frac{x_k}{\sqrt{u_k}} \\ &\leq \{\text{Equation (3.50)}\} \\ &\quad \frac{x_{i^*}}{\sqrt{u_{i^*}}} + \frac{x_j}{\sqrt{u_j}} + \sum_{\tau_k \in \tau_{\text{act}}(t) \setminus \{\tau_{i^*}, \tau_j\}} \frac{x_k}{\sqrt{u_k}} \\ &= \sum_{\tau_k \in \tau_{\text{act}}(t)} \frac{x_k}{\sqrt{u_k}}. \end{aligned}$$ $\blacksquare$



(a) Pythagorean Theorem.



(b) Triangle Inequality.

Figure 3.12: Triangle properties.

Observe that any solution $\overrightarrow{\mathbf{x}}$ that is not of form $x_{i^*} = \sqrt{\beta_{\text{Unr}}}$ such that $u_{i^*} = \max_{\tau_k \in \tau_{\text{act}}(t)} \{u_k\}$ and $x_j = 0$ for any $j \neq i^*$ can be modified as described by Claim 3.33.2 without increasing the objective value, thereby increasing x_{i^*} and setting some x_j to 0. Thus, there must be an optimal solution of this form.

The objective value of such an optimal solution is then

$$\begin{aligned}
\sum_{\tau_k \in \tau_{\text{act}}(t)} \frac{x_k}{\sqrt{u_k}} &= \{j \neq i \Rightarrow x_j = 0\} \\
&\quad \frac{x_{i^*}}{\sqrt{u_{i^*}}} \\
&= \frac{\sqrt{\beta_{\text{Unr}}}}{\sqrt{u_{i^*}}} \\
&= \{\text{Definition 3.10}\} \\
&\quad \frac{\sqrt{u_{[1]}} \cdot \frac{(n_{\text{act}})\left(\phi + T_{[1]}\right)\left(sp^{\max} + u_{[n]}\right)}{s\ell \cdot u_{[n]}}}{\sqrt{u_{i^*}}} \\
&\geq \frac{(n_{\text{act}})\left(\phi + T_{[1]}\right)\left(sp^{\max} + u_{[n]}\right)}{s\ell \cdot u_{[n]}}.
\end{aligned}$$

□

▷ **Lemma 3.34.** Let $[t_0, t_1)$ be a time interval such that for any $t \in [t_0, t_1)$, τ has slowdown factor $s\ell$ and

$$\exists \tau^{\text{const}} \subseteq \tau : \ \forall t \in [t_0, t_1) : \ \tau_{\text{act}}(t) = \tau^{\text{const}}$$

and at time t_0, we have

$$\sum_{\tau_i \in \tau^{\text{const}}} \left(dev_i(t_0)\right)^2 \leq \beta_{\text{Unr}}. \tag{3.51}$$

We have

$$\sum_{\tau_i \in \tau^{\text{const}}} \left(dev_i(t)\right)^2 \leq \beta_{\text{Unr}}. \tag{3.52}$$

for any $t \in [t_0, t_1)$. ◁

Proof. We prove the lemma by contradiction. Suppose otherwise that there exists at least one time instant in $[t_0, t_1)$ such that (3.52) is false. By (3.51), (3.52) is true at time t_0. Let $t_{\text{b}} \in [t_0, t_1)$ denote the latest time instant such that (3.52) is true over $[t_0, t_{\text{b}})$. We will show that the existence of t_{b} leads to a contradiction.

► **Claim 3.34.1.** $\sum_{\tau_i \in \tau^{\text{const}}} \left(dev_i(t_{\text{b}})\right)^2 = \beta_{\text{Unr}}$. ◄

Proof. By the definition of t_{b}, we have that

$$\forall t \in [t_0, t_b) : \sum_{\tau_i \in \tau^{\text{const}}} \left(dev_i(t)\right)^2 \leq \beta_{\text{Unr}}. \tag{3.53}$$

Thus,

$$\begin{aligned}
\beta_{\text{Unr}} \geq\ & \{\text{Expression (3.53)}\} \\
& \lim_{t^* \to t_{\text{b}}^-} \sum_{\tau_i \in \tau^{\text{const}}} \left(dev_i(t^*)\right)^2 \\
=\ & \sum_{\tau_i \in \tau^{\text{const}}} \left(\lim_{t^* \to t_{\text{b}}^-} dev_i(t^*) \right)^2 \\
\geq\ & \left\{ \text{By Definition 3.2, } dev_i(t_{\text{b}}) \geq 0, \text{ and, by Lemma 3.10, } \lim_{t^* \to t_{\text{b}}^-} dev_i(t^*) \geq dev_i(t_{\text{b}}) \right\} \\
& \sum_{\tau_i \in \tau^{\text{const}}} \left(dev_i(t_{\text{b}})\right)^2.
\end{aligned}$$

Also by the definition of t_{b}, we have that

$$\forall \delta > 0 : \exists t \in [t_{\mathrm{b}}, t_{\mathrm{b}} + \delta) : \sum_{\tau_i \in \tau^{\text{const}}} (dev_i(t))^2 > \beta_{\text{Unr}}, \tag{3.54}$$

because otherwise t_{b} is not the latest time instant such that (3.52) is true over $[t_0, t_{\mathrm{b}})$. Thus,

$$\begin{aligned}
\beta_{\text{Unr}} \leq\ & \{\text{Equation (3.54)}\} \\
& \lim_{t^* \to t_{\mathrm{b}}^+} \sum_{\tau_i \in \tau^{\text{const}}} (dev_i(t^*))^2 \\
=\ & \sum_{\tau_i \in \tau^{\text{const}}} \left(\lim_{t^* \to t_{\mathrm{b}}^+} dev_i(t^*) \right)^2 \\
=\ & \{\text{Corollary 3.12}\} \\
& \sum_{\tau_i \in \tau^{\text{const}}} (dev_i(t_{\mathrm{b}}))^2 .
\end{aligned}$$

Because $\beta_{\text{Unr}} \leq \sum_{\tau_i \in \tau^{\text{const}}} (dev_i(t_{\mathrm{b}}))^2 \leq \beta_{\text{Unr}}$, we must have $\sum_{\tau_i \in \tau^{\text{const}}} (dev_i(t_{\mathrm{b}}))^2 = \beta_{\text{Unr}}$. ■

► **Claim 3.34.2.** $\sum_{\tau_i \in \tau^{\text{const}}} \frac{dev_i(t_{\mathrm{b}})}{\sqrt{u_i}} \geq \frac{(n_{\text{act}})(\phi + T_{[1]})(sp^{\max} + u_{[n]})}{s\ell \cdot u_{[n]}}$. ◄

Proof. Consider, for each task $\tau_i \in \tau^{\text{const}}$, letting $x_i = dev_i(t_{\mathrm{b}})$ in the optimization problem presented in Lemma 3.33. By Claim 3.34.1, letting $x_i = dev_i(t_{\mathrm{b}})$ satisfies constraint (3.47). By Definition 3.2, each $dev_i(t_{\mathrm{b}}) \geq 0$. Thus, letting $x_i = dev_i(t_{\mathrm{b}})$ also satisfies constraint (3.48). The claim follows from Lemma 3.33. ■

► **Claim 3.34.3.** $\sum_{\tau_i \in \tau} dev_i(t_{\mathrm{b}}) \cdot \left(\frac{csp_i(t_{\mathrm{b}})}{\sqrt{u_i}} - \sqrt{u_i} \right) > 0$. ◄

Proof. Let $\mathbf{X}^{s\ell} \in \mathbb{R}_{\geq 0}^{n \cdot m}$ be a solution to constraints (3.39)-(3.41) in Definition 3.8 at time t_{b}. Such an $\mathbf{X}^{s\ell}$ is guaranteed to exist by the lemma statement (*i.e.*, τ has slowdown factor $s\ell$) and Definition 3.8.

Let $\mathbf{X}^{\text{cfg}}$ denote an optimal solution to the AP instance defined by Unr-WC at t_{b} that corresponds with the configuration chosen by Unr-WC at time t_{b}, *i.e.*, ready task τ_i is scheduled on processor π_j only if $x_{i,j}^{\text{cfg}} = 1$.

Compare constraints (3.40) and (3.41) as mentioned in Definition 3.8 with constraints (2.2) and (2.3) of the AP instance. Observe that because $\mathbf{X}^{s\ell}$ satisfies (3.40) and (3.41), $\mathbf{X}^{s\ell}$ also satisfies

constraints (2.2) and (2.3) of the AP instance. The only constraint of the AP instance that $\mathbf{X}^{s\ell}$ may not satisfy is (2.4). By Theorem 2.1, the objective function value for integral optimal solution $\mathbf{X}^{\text{cfg}}$ is at least the value for non-integral solution $\mathbf{X}^{s\ell}$. By the definition of Unr-WC, the objective function value of the AP instance for arbitrary solution $\mathbf{X}$ is $\sum_{\tau_i \in \tau} \sum_{\pi_j \in \pi} \Psi_i(t_{\text{b}}) \cdot sp^{i,j} \cdot x_{i,j}$. Thus,

$$\sum_{\tau_i \in \tau} \sum_{\pi_j \in \pi} \Psi_i(t_{\text{b}}) \cdot sp^{i,j} \cdot x_{i,j}^{\text{cfg}} \geq \sum_{\tau_i \in \tau} \sum_{\pi_j \in \pi} \Psi_i(t_{\text{b}}) \cdot sp^{i,j} \cdot x_{i,j}^{s\ell}. \tag{3.55}$$

We have

$$\left.\begin{aligned}
&\sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}})} \Psi_i(t_{\text{b}}) \cdot csp_i(t_{\text{b}}) \\
=& \sum_{\tau_i \in \tau_{\text{rdy}}(t_{\text{b}})} \Psi_i(t_{\text{b}}) \cdot csp_i(t_{\text{b}}) + \sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}}) \setminus \tau_{\text{rdy}}(t_{\text{b}})} \Psi_i(t_{\text{b}}) \cdot csp_i(t_{\text{b}}) \\
=& \left\{\text{By Definition 3.6, } \tau_i \notin \tau_{\text{rdy}}(t_{\text{b}}) \Rightarrow \Psi_i(t_{\text{b}}) = 0\right\} \\
& \sum_{\tau_i \in \tau_{\text{rdy}}(t_{\text{b}})} \Psi_i(t_{\text{b}}) \cdot csp_i(t_{\text{b}}) + \sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}}) \setminus \tau_{\text{rdy}}(t_{\text{b}})} 0 \\
=& \sum_{\tau_i \in \tau_{\text{rdy}}(t_{\text{b}})} \Psi_i(t_{\text{b}}) \cdot csp_i(t_{\text{b}}) + \sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}}) \setminus \tau_{\text{rdy}}(t_{\text{b}})} \sum_{\pi_j \in \pi} 0 \cdot sp^{i,j} \cdot x_{i,j}^{\text{cfg}} \\
=& \left\{\text{By Definition 3.6, } \tau_i \notin \tau_{\text{rdy}}(t_{\text{b}}) \Rightarrow \Psi_i(t_{\text{b}}) = 0\right\} \\
& \sum_{\tau_i \in \tau_{\text{rdy}}(t_{\text{b}})} \Psi_i(t_{\text{b}}) \cdot csp_i(t_{\text{b}}) + \sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}}) \setminus \tau_{\text{rdy}}(t_{\text{b}})} \sum_{\pi_j \in \pi} \Psi_i(t_{\text{b}}) \cdot sp^{i,j} \cdot x_{i,j}^{\text{cfg}}.
\end{aligned}\right\} \tag{3.56}$$

By Definition 2.21, if, for task τ_i, we have $csp_i(t_{\text{b}}) \neq 0$ (*i.e.*, task τ_i's execution speed is nonzero at time t_{b}), then τ_i is scheduled on some processor π_{j^*} with speed sp^{i,j^*} at time t_{b} and $csp_i(t_{\text{b}}) = sp^{i,j^*}$. If task τ_i is scheduled on processor π_{j^*}, by the definitions of Unr-WC and $\mathbf{X}^{\text{cfg}}$, we have $x_{i,j^*}^{\text{cfg}} = 1$ and, for any $\pi_j \neq \pi_{j^*}$, $x_{i,j}^{\text{cfg}} = 0$. Thus,

$$csp_i(t_{\text{b}}) \neq 0 \Rightarrow csp_i(t_{\text{b}}) = \sum_{\pi_j \in \pi} sp^{i,j} \cdot x_{i,j}^{\text{cfg}}. \tag{3.57}$$

Continuing the derivation in (3.56), we have

$$\left.\begin{aligned}
&\sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}})} \Psi_i(t_{\text{b}}) \cdot csp_i(t_{\text{b}}) \\
&= \sum_{\tau_i \in \tau_{\text{rdy}}(t_{\text{b}})} \Psi_i(t_{\text{b}}) \cdot csp_i(t_{\text{b}}) + \sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}}) \setminus \tau_{\text{rdy}}(t_{\text{b}})} \sum_{\pi_j \in \pi} \Psi_i(t_{\text{b}}) \cdot sp^{i,j} \cdot x_{i,j}^{\text{cfg}} \\
&= \{\text{Equation (3.57)}\} \\
&\quad \sum_{\tau_i \in \tau_{\text{rdy}}(t_{\text{b}})} \sum_{\pi_j \in \pi} \Psi_i(t_{\text{b}}) \cdot sp^{i,j} \cdot x_{i,j}^{\text{cfg}} + \sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}}) \setminus \tau_{\text{rdy}}(t_{\text{b}})} \sum_{\pi_j \in \pi} \Psi_i(t_{\text{b}}) \cdot sp^{i,j} \cdot x_{i,j}^{\text{cfg}} \\
&= \sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}})} \sum_{\pi_j \in \pi} \Psi_i(t_{\text{b}}) \cdot sp^{i,j} \cdot x_{i,j}^{\text{cfg}} \\
&= \left\{ \begin{array}{l} \text{By Definitions 2.11 and 2.13, } \tau_i \notin \tau_{\text{act}}(t_{\text{b}}) \Rightarrow \tau_i \notin \tau_{\text{rdy}}(t_{\text{b}}) \text{ and,} \\ \text{by Definition 3.6, } \tau_i \notin \tau_{\text{rdy}}(t_{\text{b}}) \Rightarrow \Psi_i(t_{\text{b}}) = 0 \end{array} \right\} \\
&\quad \sum_{\tau_i \in \tau} \sum_{\pi_j \in \pi} \Psi_i(t_{\text{b}}) \cdot sp^{i,j} \cdot x_{i,j}^{\text{cfg}} \\
&\geq \{\text{Equation (3.55)}\} \\
&\quad \sum_{\tau_i \in \tau} \sum_{\pi_j \in \pi} \Psi_i(t_{\text{b}}) \cdot sp^{i,j} \cdot x_{i,j}^{s\ell} \\
&= \left\{ \begin{array}{l} \text{By Definitions 2.11 and 2.13, } \tau_i \notin \tau_{\text{act}}(t_{\text{b}}) \Rightarrow \tau_i \notin \tau_{\text{rdy}}(t_{\text{b}}) \text{ and,} \\ \text{by Definition 3.6, } \tau_i \notin \tau_{\text{rdy}}(t_{\text{b}}) \Rightarrow \Psi_i(t_{\text{b}}) = 0 \end{array} \right\} \\
&\quad \sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}})} \sum_{\pi_j \in \pi} \Psi_i(t_{\text{b}}) \cdot sp^{i,j} \cdot x_{i,j}^{s\ell}.
\end{aligned}\right\} \tag{3.58}$$

By the definition of $\mathbf{X}^{s\ell}$, $\mathbf{X}^{s\ell}$ satisfies (3.39) at time t_{b}. By multiplying both sides of (3.39) by $\frac{\Psi_i(t_{\text{b}})}{1.0 - s\ell}$ and summing over the tasks in $\tau_{\text{act}}(t_{\text{b}})$ (note that this preserves the direction of the inequality because, by Definitions 3.6 and 3.8, $\frac{\Psi_i(t_{\text{b}})}{1.0 - s\ell} \geq 0$ for each task τ_i), we have $\sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}})} \sum_{\pi_j \in \pi} \Psi_i(t_{\text{b}}) \cdot sp^{i,j} \cdot x_{i,j}^{s\ell} \geq \sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}})} \Psi_i(t_{\text{b}}) \frac{u_i}{1.0 - s\ell}$. By (3.58), we have

$$\sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}})} \Psi_i(t_{\text{b}}) \cdot csp_i(t_{\text{b}}) \geq \sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}})} \Psi_i(t_{\text{b}}) \frac{u_i}{1.0 - s\ell}. \tag{3.59}$$

Thus,

$$\sum_{\tau_i \in \tau} dev_i(t_\text{b}) \cdot \left(\frac{csp_i(t_\text{b})}{\sqrt{u_i}} - \sqrt{u_i} \right)$$

$= \{\text{Lemma 3.5}\}$

$$\sum_{\tau_i \in \tau_\text{act}(t_\text{b})} dev_i(t_\text{b}) \cdot \left(\frac{csp_i(t_\text{b})}{\sqrt{u_i}} - \sqrt{u_i} \right)$$

$$= \sum_{\tau_i \in \tau_\text{act}(t_\text{b})} \frac{dev_i(t_\text{b})}{\sqrt{u_i}} \cdot (csp_i(t_\text{b}) - u_i)$$

$$\geq \left\{ \begin{array}{l} \text{By Lemma 3.32, } \left| \frac{dev_i(t_\text{b})}{\sqrt{u_i}} - \Psi_i(t_\text{b}) \right| \leq T_{[1]} + \phi \\ \Rightarrow \Psi_i(t_\text{b}) - \frac{dev_i(t_\text{b})}{\sqrt{u_i}} \leq T_{[1]} + \phi \\ \Rightarrow \Psi_i(t_\text{b}) - T_{[1]} - \phi \leq \frac{dev_i(t_\text{b})}{\sqrt{u_i}} \end{array} \right\}$$

$$\sum_{\tau_i \in \tau_\text{act}(t_\text{b})} \left(\Psi_i(t_\text{b}) - T_{[1]} - \phi \right) (csp_i(t_\text{b}) - u_i)$$

$$= \sum_{\tau_i \in \tau_\text{act}(t_\text{b})} \Psi_i(t_\text{b}) \cdot csp_i(t) - \sum_{\tau_i \in \tau_\text{act}(t_\text{b})} \Psi_i(t_\text{b}) \cdot u_i - \sum_{\tau_i \in \tau_\text{act}(t_\text{b})} \left(T_{[1]} + \phi \right) (csp_i(t_\text{b}) - u_i)$$

$> \{-u_i < 0 \text{ and, by Definitions 2.21 and 3.9, } sp^\text{max} \geq csp_i(t_\text{b})\}$

$$\sum_{\tau_i \in \tau_\text{act}(t_\text{b})} \Psi_i(t_\text{b}) \cdot csp_i(t) - \sum_{\tau_i \in \tau_\text{act}(t_\text{b})} \Psi_i(t_\text{b}) \cdot u_i - \sum_{\tau_i \in \tau_\text{act}(t_\text{b})} \left(T_{[1]} + \phi \right) (sp^\text{max})$$

$\geq \{\text{By Definition 2.13, } n_\text{act} \geq |\tau_\text{act}(t_\text{b})|\}$

$$\sum_{\tau_i \in \tau_\text{act}(t_\text{b})} \Psi_i(t_\text{b}) \cdot csp_i(t) - \sum_{\tau_i \in \tau_\text{act}(t_\text{b})} \Psi_i(t_\text{b}) \cdot u_i - (n_\text{act}) \left(T_{[1]} + \phi \right) (sp^\text{max})$$

$$= \sum_{\tau_i \in \tau_\text{act}(t_\text{b})} \Psi_i(t_\text{b}) \cdot csp_i(t) - \sum_{\tau_i \in \tau_\text{act}(t_\text{b})} \Psi_i(t_\text{b}) \frac{u_i}{1.0 - s\ell} + \sum_{\tau_i \in \tau_\text{act}(t_\text{b})} \Psi_i(t_\text{b}) \frac{u_i}{1.0 - s\ell}$$
$$- \sum_{\tau_i \in \tau_\text{act}(t_\text{b})} \Psi_i(t_\text{b}) \cdot u_i - (n_\text{act}) \left(T_{[1]} + \phi \right) (sp^\text{max})$$

$\geq \{\text{Equation (3.59)}\}$

$$\sum_{\tau_i \in \tau_\text{act}(t_\text{b})} \Psi_i(t_\text{b}) \frac{u_i}{1.0 - s\ell} - \sum_{\tau_i \in \tau_\text{act}(t_\text{b})} \Psi_i(t_\text{b}) \cdot u_i - (n_\text{act}) \left(T_{[1]} + \phi \right) (sp^\text{max})$$

$$= \left\{ \frac{1}{1.0 - s\ell} = \sum_{k=0}^{\infty} s\ell^k \text{ because, by Definition 3.8, } s\ell \in (0, 1) \right\}$$

$$\sum_{\tau_i \in \tau_\text{act}(t_\text{b})} \Psi_i(t_\text{b}) \cdot u_i \left(\sum_{k=0}^{\infty} s\ell^k \right) - \sum_{\tau_i \in \tau_\text{act}(t_\text{b})} \Psi_i(t_\text{b}) \cdot u_i - (n_\text{act}) \left(T_{[1]} + \phi \right) (sp^\text{max})$$

$$
\begin{aligned}
&= \sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}})} \Psi_i(t_{\text{b}}) \cdot u_i \left(\sum_{k=1}^{\infty} s\ell^k \right) - (n_{\text{act}}) \left(T_{[1]} + \phi \right) (sp^{\max}) \\
&\geq \sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}})} \Psi_i(t_{\text{b}}) \cdot u_i \cdot s\ell - (n_{\text{act}}) \left(T_{[1]} + \phi \right) (sp^{\max}) \\
&\geq \left\{ u_i \geq u_{[n]} \right\} \\
&\quad \sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}})} \Psi_i(t_{\text{b}}) \cdot u_{[n]} \cdot s\ell - (n_{\text{act}}) \left(T_{[1]} + \phi \right) (sp^{\max}) \\
&\geq \left\{ \text{By Lemma 3.32, } \left| \frac{dev_i(t_{\text{b}})}{\sqrt{u_i}} - \Psi_i(t_{\text{b}}) \right| \leq T_{[1]} + \phi \Rightarrow \frac{dev_i(t_{\text{b}})}{\sqrt{u_i}} - T_{[1]} - \phi \leq \Psi_i(t_{\text{b}}) \right\} \\
&\quad \sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}})} \left(\frac{dev_i(t_{\text{b}})}{\sqrt{u_i}} - T_{[1]} - \phi \right) \cdot u_{[n]} \cdot s\ell - (n_{\text{act}}) \left(T_{[1]} + \phi \right) (sp^{\max}) \\
&= \sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}})} \left(\frac{dev_i(t_{\text{b}})}{\sqrt{u_i}} \right) \cdot u_{[n]} \cdot s\ell - \sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}})} \left(T_{[1]} + \phi \right) \cdot u_{[n]} \cdot s\ell \\
&\quad - (n_{\text{act}}) \left(T_{[1]} + \phi \right) (sp^{\max}) \\
&> \left\{ s\ell \in (0,1) \Rightarrow -u_{[n]} \cdot s\ell > -u_{[n]} \right\} \\
&\quad \sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}})} \left(\frac{dev_i(t_{\text{b}})}{\sqrt{u_i}} \right) \cdot u_{[n]} \cdot s\ell - \sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}})} \left(T_{[1]} + \phi \right) u_{[n]} \\
&\quad - (n_{\text{act}}) \left(T_{[1]} + \phi \right) (sp^{\max}) \\
&\geq \left\{ \text{By Definition 2.13, } n_{\text{act}} \geq |\tau_{\text{act}}(t_{\text{b}})| \right\} \\
&\quad \sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}})} \left(\frac{dev_i(t_{\text{b}})}{\sqrt{u_i}} \right) \cdot u_{[n]} \cdot s\ell - (n_{\text{act}}) \left(T_{[1]} + \phi \right) u_{[n]} - (n_{\text{act}}) \left(T_{[1]} + \phi \right) (sp^{\max}) \\
&= \sum_{\tau_i \in \tau_{\text{act}}(t_{\text{b}})} \left(\frac{dev_i(t_{\text{b}})}{\sqrt{u_i}} \right) \cdot u_{[n]} \cdot s\ell - (n_{\text{act}}) \left(T_{[1]} + \phi \right) \left(sp^{\max} + u_{[n]} \right) \\
&\geq \left\{ \text{Claim 3.34.2} \right\} \\
&\quad (n_{\text{act}}) \left(\phi + T_{[1]} \right) \left(sp^{\max} + u_{[n]} \right) - (n_{\text{act}}) \left(T_{[1]} + \phi \right) \left(sp^{\max} + u_{[n]} \right) \\
&= 0.
\end{aligned}
$$

This is the claim. ■

By Claim 3.34.3 and Lemma 3.31, we have $\exists t^* \in (t_{\text{b}}, t_1) : \forall t \in [t_{\text{b}}, t^*) : \sum_{\tau_i \in \tau} (dev_i(t))^2 \leq \sum_{\tau_i \in \tau} (dev_i(t_{\text{b}}))^2$. Because $\tau_{\text{act}}(t) = \tau^{\text{const}}$ for $t \in [t_0, t_1)$ and by Lemma 3.5, we have $\exists t^* \in (t_{\text{b}}, t_1) : \forall t \in [t_{\text{b}}, t^*) : \sum_{\tau_i \in \tau^{\text{const}}} (dev_i(t))^2 \leq \sum_{\tau_i \in \tau^{\text{const}}} (dev_i(t_{\text{b}}))^2$. By Claim 3.34.1, we have

$\exists t^* \in (t_\mathrm{b}, t_1) : \forall t \in [t_\mathrm{b}, t^*) : \sum_{\tau_i \in \tau^{\text{const}}} (dev_i(t))^2 \leq \beta_{\text{Unr}}$. This contradicts that t_b is the latest time instant such that (3.52) is true over $[t_0, t_\mathrm{b})$, proving the lemma. □

Dynamic tasks. We now show that the sum of squares of deviations is upper bounded even when tasks enter and leave the system.

▷ **Lemma 3.35.** If task system τ has slowdown factor $s\ell$, then we have

$$\sum_{\tau_i \in \tau_{\text{act}}(t)} (dev_i(t))^2 \leq \beta_{\text{Unr}}$$

for any time t. ◁

Proof. We prove by induction on the activation time instants t_k^{act} for $k \in \mathbb{N}$. The induction hypothesis is as follows:

$$\forall t \in (-\infty, t_k^{\text{act}}] : \sum_{\tau_i \in \tau_{\text{act}}(t)} (dev_i(t))^2 \leq \beta_{\text{Unr}}. \tag{3.60}$$

The base case of $k = 1$ is considered by the following claim.

► **Claim 3.35.1.** $\forall t \in (-\infty, t_1^{\text{act}}] : \sum_{\tau_i \in \tau_{\text{act}}(t)} (dev_i(t))^2 \leq \beta_{\text{Unr}}$. ◄

Proof. By Definition 2.14, $\forall t \in (-\infty, t_1^{\text{act}})$,

$$\begin{aligned}\sum_{\tau_i \in \tau_{\text{act}}(t)} (dev_i(t))^2 &= \sum_{\tau_i \in \emptyset} (dev_i(t))^2 \\ &= 0 \\ &\leq \beta_{\text{Unr}}.\end{aligned}$$

It remains to prove that $\sum_{\tau_i \in \tau_{\text{act}}(t_1^{\text{act}})} (dev_i(t_1^{\text{act}}))^2 \leq \beta_{\text{Unr}}$. We have

$$\begin{aligned}\sum_{\tau_i \in \tau_{\text{act}}(t_1^{\text{act}})} \left(dev_i(t_1^{\text{act}})\right)^2 &\leq \{\text{Lemma 3.10}\} \\ &\sum_{\tau_i \in \tau_{\text{act}}(t_1^{\text{act}})} \left(\lim_{t^* \to (t_1^{\text{act}})^-} dev_i(t^*) \right)^2.\end{aligned} \tag{3.61}$$

By Definition 2.14, for any task τ_i and $t^* < t_1^{\text{act}}$, task τ_i is inactive at t^*. By Lemma 3.5, for any time $t^* < t_1^{\text{act}}$, $dev_i(t^*) = 0$. Thus, $\lim_{t^* \to (t_1^{\text{act}})^-} dev_i(t^*) = 0$. Continuing from the derivation paused at (3.61), we have

$$\begin{aligned}
\sum_{\tau_i \in \tau_{\text{act}}(t_1^{\text{act}})} \left(dev_i(t_1^{\text{act}})\right)^2 &\leq \sum_{\tau_i \in \tau_{\text{act}}(t_1^{\text{act}})} \left(\lim_{t^* \to (t_1^{\text{act}})^-} dev_i(t^*) \right)^2 \\
&= \sum_{\tau_i \in \tau_{\text{act}}(t_1^{\text{act}})} (0)^2 \\
&= 0 \\
&\leq \beta_{\text{Unr}}.
\end{aligned}$$

This concludes the proof of Claim 3.35.1, the base case of the inductive proof of Lemma 3.35. ■

Our remaining obligation is to prove that (3.60) implies the $(k+1)^{\text{th}}$ case. This is split among the following two claims.

► **Claim 3.35.2.** (3.60) implies that

$$\forall t \in [t_k^{\text{act}}, t_{k+1}^{\text{act}}) : \sum_{\tau_i \in \tau_{\text{act}}(t_k^{\text{act}})} (dev_i(t))^2 \leq \beta_{\text{Unr}}.$$ ◄

Proof. Let $\tau^{\text{const}} \triangleq \tau_{\text{act}}(t_k^{\text{act}})$. By Definition 2.14, $\forall t \in [t_k^{\text{act}}, t_{k+1}^{\text{act}}) : \tau_{\text{act}}(t) = \tau^{\text{const}}$. The claim follows from (3.60) and Lemma 3.34 with $[t_0, t_1) = [t_k^{\text{act}}, t_{k+1}^{\text{act}})$. ■

► **Claim 3.35.3.** $\sum_{\tau_i \in \tau_{\text{act}}(t_{k+1}^{\text{act}})} \left(dev_i(t_{k+1}^{\text{act}})\right)^2 \leq \beta_{\text{Unr}}$. ◄

Proof. Let $\tau^{\text{old}} \triangleq \tau_{\text{act}}(t_{k+1}^{\text{act}}) \cap \tau_{\text{act}}(t_k^{\text{act}})$ and $\tau^{\text{new}} \triangleq \tau_{\text{act}}(t_{k+1}^{\text{act}}) \setminus \tau_{\text{act}}(t_k^{\text{act}})$. τ^{old} denotes tasks of $\tau_{\text{act}}(t_{k+1}^{\text{act}})$ that were also active in $[t_k^{\text{act}}, t_{k+1}^{\text{act}})$, while τ^{new} denotes tasks of $\tau_{\text{act}}(t_{k+1}^{\text{act}})$ that became active at time t_{k+1}^{act}.

We have

$$
\begin{aligned}
&\sum_{\tau_i \in \tau_{\text{act}}\left(t_{k+1}^{\text{act}}\right)} \left(dev_i\left(t_{k+1}^{\text{act}}\right)\right)^2 \\
&= \left[\sum_{\tau_i \in \tau^{\text{old}}} \left(dev_i\left(t_{k+1}^{\text{act}}\right)\right)^2\right] + \left[\sum_{\tau_i \in \tau^{\text{new}}} \left(dev_i\left(t_{k+1}^{\text{act}}\right)\right)^2\right] \\
&\leq \{\text{Lemma 3.10}\} \\
&\quad \left[\sum_{\tau_i \in \tau^{\text{old}}} \left(\lim_{t^* \to \left(t_{k+1}^{\text{act}}\right)^-} dev_i(t^*)\right)^2\right] + \left[\sum_{\tau_i \in \tau^{\text{new}}} \left(\lim_{t^* \to \left(t_{k+1}^{\text{act}}\right)^-} dev_i(t^*)\right)^2\right] \\
&\leq \left\{\text{Claim 3.35.2 and } \tau^{\text{old}} \subseteq \tau_{\text{act}}\left(t_k^{\text{act}}\right)\right\} \\
&\quad \beta_{\text{Unr}} + \left[\sum_{\tau_i \in \tau^{\text{new}}} \left(\lim_{t^* \to \left(t_{k+1}^{\text{act}}\right)^-} dev_i(t^*)\right)\right] \\
&= \left\{\text{Lemma 3.5 and } \tau_i \in \tau^{\text{new}} \Rightarrow \tau_i \text{ inactive over } [t_k^{\text{act}}, t_{k+1}^{\text{act}})\right\} \\
&\quad \beta_{\text{Unr}} + 0.
\end{aligned}
$$

This completes the proof of the claim. ■

Claims 3.35.2 and 3.35.3 form the induction step, thereby proving the induction hypothesis (3.60) for any $k \in \mathbb{N}$. Taking $k \to \infty$ yields the lemma statement. □

Theorem 3.36 presents our response-time bound for Unr-WC under UNRELATED.

▷ **Theorem 3.36.** For any task system τ on UNRELATED with slowdown factor $s\ell$, the response time of any task τ_i when scheduled under Ufm-WC is at most

$$T_i + \sqrt{\frac{u_{[1]}}{u_i}} \frac{(n_{\text{act}})\left(T_{[1]} + \phi\right)\left(sp^{\max} + u_{[n]}\right)}{u_{[n]} \cdot s\ell}. \qquad \triangleleft$$

Proof. Consider $dev_i(t)$ at any time instant t. There are two cases.

◄ **Case 3.36.1.** Task τ_i is inactive at t. ►

By Lemma 3.5, $dev_i(t) = 0$. ♦

◄ **Case 3.36.2.** Task τ_i is active at t. ►

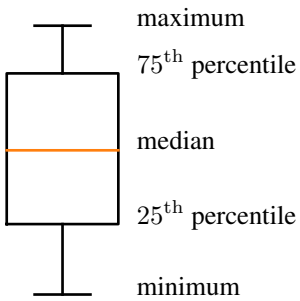


Figure 3.13: Interpreting a boxplot.

By Definition 2.13, $\tau_i \in \tau_{\text{act}}(t)$. By Lemma 3.35, we have $\sum_{\tau_k \in \tau_{\text{act}}(t)} (dev_k(t))^2 \leq \beta_{\text{Unr}}$. Because squares are non-negative, we have $(dev_i(t))^2 \leq \beta_{\text{Unr}}$. By Definition 3.10, we have

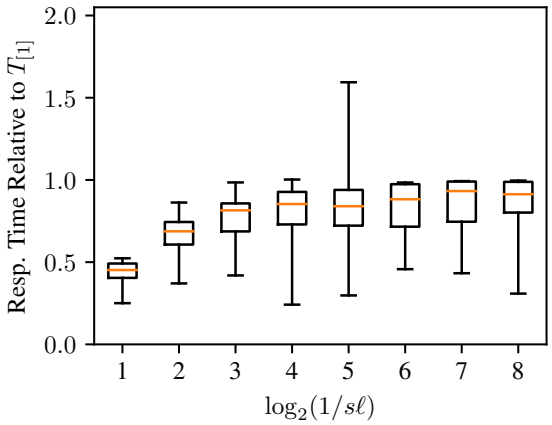
$$dev_i(t) \leq \sqrt{u_{[n]}} \frac{(n_{\text{act}}) \left(T_{[1]} + \phi\right) \left(sp^{\max} + u_{[n]}\right)}{u_{[n]} \cdot s\ell}. \qquad \blacklozenge$$

In either case, we have $dev_i(t) \leq \sqrt{u_{[n]}} \frac{(n_{\text{act}}) \left(T_{[1]} + \phi\right) \left(sp^{\max} + u_{[n]}\right)}{u_{[n]} \cdot s\ell}$. The theorem follows from Lemma 3.4. $\square$

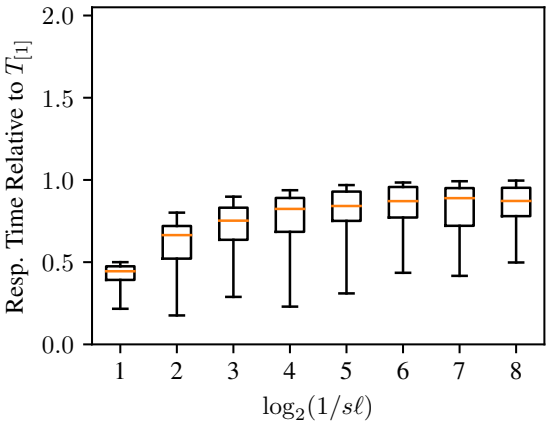
3.5.3 Evaluation

To evaluate the looseness of our response-time bound, we simulated Unr-WC with the priority point of task τ_i set as $pp_i(t) = t - \left(\left\lfloor \frac{t}{T_i} \right\rfloor + 2\right) T_i + \tilde{d}_i(t)$ (recall from Section 3.5.1.1 that this choice of priority point mitigates the non-work-conserving behavior and unpredictable migrations of Unr-WC) on randomly generated periodic task systems and multiprocessors in Python. We generated task systems of sizes $n \in \{20, 40, 80\}$, with $m \in \{4, 8\}$. We also considered values of $s\ell$ ranging from $\{1/2, 1/4, 1/8, \ldots, 1/256\}$. Processor speeds for each task were sampled uniformly from [0.0, 1.0). Utilizations were generated to match given $s\ell$ values by solving a maximization linear program with constraints taken from those of UNRELATED-Feasible with decision variables $\mathbf{X}$ and $\overrightarrow{\mathbf{u}}$. The objective function was a linear combination of the elements of $\overrightarrow{\mathbf{u}}$, with coefficients sampled uniformly from [0.0, 1.0). Periods were then sampled uniformly from [10, 100]. 100 task systems and multiprocessors were generated for each triplet of n, m, and $s\ell$ value. For each generated system, response times of tasks were measured for 100,000 simulated time units.

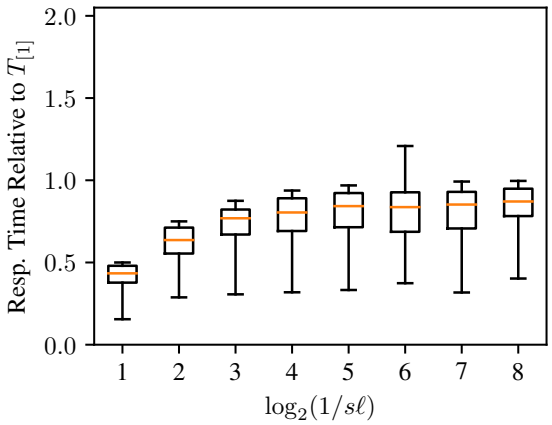
For each pair of n and m, we plotted the maximum response time relative to the maximum period $T_{[1]}$ of each task system against $s\ell$. Boxplots illustrate the distribution of maximum response times (relative to



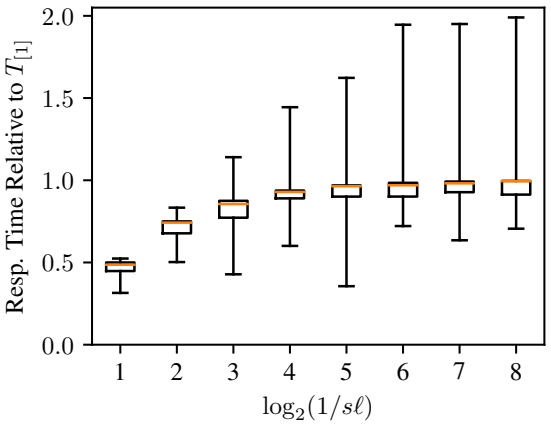
(a) $(20, 4)$.



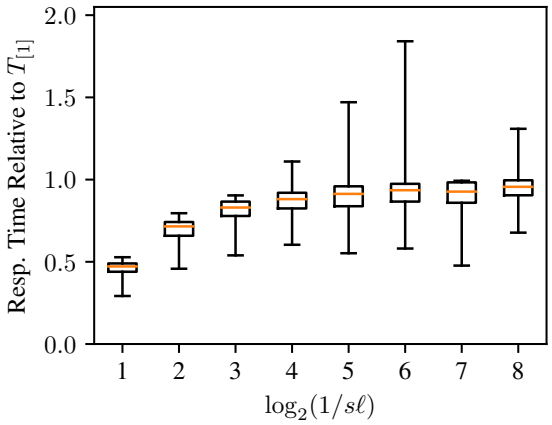
(b) $(40, 4)$.



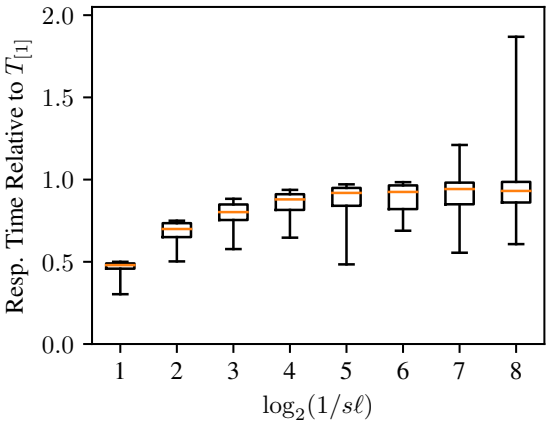
(c) $(80, 4)$.



(d) $(20, 8)$.



(e) $(40, 8)$.



(f) $(80, 8)$.

Figure 3.14: Response time against $s\ell$. Captions indicate (n, m).

each task system's $T_{[1]}$) for the 100 task systems generated for each $(n, m, s\ell)$ triple. Each boxplot shows the quartiles of the distribution, *i.e.*, the 0^{th} (minimum), 25^{th}, 50^{th} (median), 75^{th}, and 100^{th} (maximum) percentiles (see Figure 3.13).

These plots are presented in Figure 3.14 ($s\ell$ halves at each step from left to right). From Figure 3.14, it can be observed that, while response times increase as $s\ell \to 0$, they do not scale inversely with $s\ell$ (unlike our analytical bound in Theorem 3.36).

While this suggests that our analysis is fundamentally pessimistic and that Unr-WC may actually be SRT-optimal, this is not conclusive evidence. It has always been the case, even for standard EDF on IDENTICAL (Devi and Anderson, 2008), that the response times of randomly generated task systems tends to be lower than the worst-case response times of hand-crafted task systems. Unfortunately, the complexity of Unr-WC and, more generally, of tracking remaining execution requirements of jobs under UNRELATED seem to make computing schedules by hand intractable. For now, this has left simulation as our only approach for counterexample searching.

3.6 Chapter Summary

In this chapter, we proved response-time bounds for WC schedulers. We established that response-time bounds under $\mathcal{HP}$-$\mathcal{LAG}$ systems and proved that Ufm-WC under UNIFORM and Strong-APA-WC under IDENTICAL/ARBITRARY satisfy $\mathcal{HP}$-$\mathcal{LAG}$ when feasible. We defined Unr-WC, a WC scheduler variant under UNRELATED, and proved that Ufm-WC and Strong-APA-WC are special cases of Unr-WC. We proved asymptotic response-time bounds that approach infinity as the system approaches violating the UNRELATED-Feasible condition. Though we failed to prove the SRT-optimality of Unr-WC, simulation showed that observed response times do not approach infinity.

CHAPTER 4: SCHED_DEADLINE BACKGROUND

The practicality of EDF variants proposed in this dissertation were explored by implementing them on top of SCHED_DEADLINE (via patches). Discussion of these patches requires an understanding of the original SCHED_DEADLINE implementation, covered in this chapter.

The following description of SCHED_DEADLINE is based on Linux 6.7. Note that the work on SCHED_DEADLINE detailed in the following chapters predates this version. Though support for some features (*e.g.*, disabling migration, core scheduling, *etc.*) of the Linux scheduler were added since these older versions, the behavior of SCHED_DEADLINE has been mostly consistent.

This chapter will cover code of the implementation. The names of types and struct members will match the actual implementation, though only members relevant to SCHED_DEADLINE are included. Function names will also match, though their presented definitions are simplified. All synchronization code (*e.g.*, spinlocks, memory barriers, read-copy-update (RCU), *etc.*) is omitted (functions will be reasoned about as if they are atomic), as well as all code that does not affect SCHED_DEADLINE's behavior (*e.g.*, scheduling statistics). Note that, as a simplification step, the bodies of many helper functions have been transplanted into their caller's pseudocode.

4.1 User-Space API

We begin by describing how the Linux scheduler is controlled via system calls and pseudo-file systems.

ID numbers of tasks. The user-space API manages tasks via ID numbers, usually process IDs (PIDs). With POSIX threads (pthreads), the value returned by the C standard library function getpid() may not match this kernel PID value. Before discussing the user-space API, we clarify this discrepancy.

From the perspective of the kernel, what we describe as a task corresponds to a struct task_struct, to be detailed later. Each task_struct is represented in the kernel by a unique (within a PID namespace) PID number (*i.e.*, task_struct contains a member with identifier pid). System calls that modify a task (*e.g.*, change its affinity) indicate the target task by its PID, which is used by the kernel to look up a pointer to the corresponding task_struct.

`getpid()` may not return the value of this PID member for a threaded task. The `pthread` library (as implemented in the GNU C standard library) assigns each thread its own `task_struct` (and hence, PID). Each `pthread`-corresponding `task_struct` also has a thread group ID (TGID), the PID of the `task_struct` that spawned the `pthread`. For `pthreads`, `getpid()` returns this TGID. Be aware that some documentation may use the term thread ID (TID) over the potentially ambiguous PID. Function `gettid()` consistently returns the PID field.

4.1.1 Scheduling Policies

The Linux scheduler is a hierarchical scheduler composed of many different schedulers, called *scheduling policies*. These policies are

- `SCHED_DEADLINE`, which uses EDF;
- `SCHED_FIFO` and `SCHED_RR`, which use fixed-priority scheduling;
- and `SCHED_NORMAL` (default policy in Linux), `SCHED_BATCH`, and `SCHED_IDLE`, which use Linux's implementation of Earliest Eligible Virtual Deadline First (EEVDF) (Stoica and Abdel-Wahab, 1995).

Note that EEVDF was recently implemented as a replacement of Linux's Completely Fair Scheduler (CFS) and reuses much of the CFS code. As such, many identifiers in the EEVDF implementation retain names referring to CFS (*e.g.*, `fair_sched_class`, discussed in Section 4.2).

Tasks of `SCHED_DEADLINE` have statically higher priority than tasks of the fixed-priority policies, which themselves have higher priority than tasks under the EEVDF policies.

A task's policy is set via the `sched_setattr()` system call,[1] whose arguments are a target task and a pointer to a `struct sched_attr`. This `struct` contains a member indicating the desired policy, as well as corresponding scheduling parameters (*e.g.*, for `SCHED_DEADLINE`, the period, budget, and relative deadline desired for the target task). Note that because `sched_setattr()` has no C library system-call wrapper, it must be invoked in a C program using the `syscall()` function. Appendix B of the `SCHED_DEADLINE` documentation (Deadline Task Scheduling) provides an example.

[1] Scheduling policy and parameters can also be set with the `sched_setscheduler()` and `sched_setparam()` system calls, though only `sched_setattr()` can set a task's policy to `SCHED_DEADLINE`. All three system calls ultimately are serviced by the same kernel function `__sched_setscheduler()`, discussed in Section 4.3.6.

Alternatively, a task's policy can be modified from the command line using the `chrt` command. For example, we can query the properties of a task with PID 2318 as follows.

```
[root ~]# chrt -p 2318
pid 2318's current scheduling policy: SCHED_OTHER
pid 2318's current scheduling priority: 0
```

This command requires two clarifications. First, `SCHED_OTHER` is a user-space synonym for `SCHED_NORMAL`, which is defined in the kernel. Second, the "current scheduling priority" printed by this command is with respect to the fixed-priority policies `SCHED_FIFO` and `SCHED_RR`. This priority value is fixed at 0 for other policies.

The policy can be changed to `SCHED_DEADLINE` as follows.

```
[root ~]# chrt -d -T 10000 -D 170000 -P 200000 -p 0 2318
[root ~]# chrt -p 2318
pid 2318's current scheduling policy: SCHED_DEADLINE
pid 2318's current scheduling priority: 0
pid 2318's current runtime/deadline/period parameters: 10000/170000/200000
```

As units are in nanoseconds, this sets the execution time to 10µs, relative deadline to 170µs, and period to 200µs. Note that the 0 in `-p 0 2318` signifies the desired priority relative to the fixed-priority policies. Even though this argument is unused when setting a task to `SCHED_DEADLINE`, it is mandatory. Omitting this argument will instead cause the command to query the current parameters, while providing any non-0 argument here will return an error.

`SCHED_DEADLINE` flags. The `sched_attr` argument of `sched_setattr()` is also used to set scheduler flags for the target task. The set of possible flags depends on the requested policy. At time of writing, `SCHED_DEADLINE` takes three flags:

- `SCHED_FLAG_RECLAIM` that, when set, scales the task's budget consumption using GRUB (see Section 4.4.8), and when unset, scales budget consumption according to asymmetric capacities and DVFS;
- `SCHED_FLAG_DL_OVERRUN` that, when set, sends `SIGXCPU` to the task whenever its budget becomes negative;

- and SCHED_FLAG_SUGOV, which is used internally by the kernel to indicate that the task is actually a privileged schedutil governor (see Sections 4.1.6 and 4.4.9) kernel thread that runs with infinite budget and higher priority than any other SCHED_DEADLINE task.

SCHED_FLAG_RECLAIM is the only flag to affect the behavior of SCHED_DEADLINE. Thus, we will omit the latter two flags when discussing the implementation.

4.1.2 Suspending and Yielding

A task can suspend due to sleeping or blocking on some resource. These suspensions are implemented by invoking Linux system calls. For example, the nanosleep() system call arms a timer for the desired suspension duration in nanoseconds, marks the calling task as interruptible (*i.e.*, the suspension can be canceled via a signal), then calls the scheduler to unschedule the now-suspended task. On the timer firing or being interrupted, the task will be marked as runnable, and the scheduler may be called if the task has high-enough priority to preempt the running task.

Tasks can also affect the scheduler by calling the sched_yield() system call, which takes no arguments and always returns 0 in Linux. Roughly, sched_yield() is used to inform the kernel that the calling task (which must be scheduled to have called sched_yield()) is permitting other tasks of similar priority to run. The exact behavior of sched_yield() is policy-specific. We will clarify this with respect to SCHED_DEADLINE in Section 4.4.3. While sched_yield() can result in the calling task being unscheduled, the calling task remains runnable (*i.e.*, is not suspended).

4.1.3 Affinities

Tasks' CPU[2] affinities are set via the sched_setaffinity() system call and cpuset controller in the control group (cgroup) pseudo-file system (usually mounted at /sys/fs/cgroup). sched_setaffinity() takes a target task PID, a bitmask size (in bytes), and a bitmask representing the desired affinity. Alternatively, taskset internally calls sched_setaffinity() to set affinities from the command line. The affinity of a task with PID 2318 can be queried as follows.

```
[root ~]# taskset -c -p 2318
pid 2318's current affinity list: 0-7
```

[2]Note that the term CPU in the Linux kernel code is equivalent to the term "processor" as used elsewhere in this dissertation.

Argument `-c` indicates that output should be returned in CPU list format. Otherwise, `taskset`'s response is a hexadecimal bitmask. The affinity can be set to CPUs 3 and 4 as follows.

```
[root ~]# taskset -c -p 3-4 2318
pid 2318's current affinity list: 0-7
pid 2318's new affinity list: 3,4
```

Note that the kernel will automatically reduce the affinity requested by `sched_setaffinity()` (and hence, the `taskset` command) to a subset of the CPUs permitted by the task's `cpuset`, detailed in the following paragraphs.

`cgroup`s allocate resources (*e.g.*, CPUs) to groups. Different resources are allocated by specific controllers (*e.g.*, the `cpuset` controller manages CPUs[3] and memory nodes). Groups are hierarchical in that all groups are descended from a default root group managing all resources. Initially, all tasks belong to this root group.

The root group is represented by a directory in the `cgroup` file system. Child groups are created by making directories in their parents' directories and may only be allocated resources owned by their parents. Child groups of the same parent may share resources. The resources and tasks owned by a group are set by writing to specific files in the corresponding directory. An example of this will be presented for the `cpuset` controller.

`cpuset`s have significance beyond setting the affinities of groups of tasks rather than individually. Based on how `cpuset`s are configured, Linux partitions the CPUs into *root domains* (*i.e.*, clusters in real-time terminology). As with the usual motivations for clustered scheduling, Linux uses root domains to reduce overheads associated with migrations between many CPUs. We will detail in later sections how unexpected scheduling behavior results when tasks' affinity masks do not match their root domains.

The interface and behavior of the `cpuset` controller changes depending on whether a version 1 or 2 `cgroup` file system is mounted. For version 1, the different controllers have distinct hierarchies. The root group for the `cpuset` controller is typically at `/sys/fs/cgroup/cpuset`. The files of interest to us are

- `cgroup.procs`, a list of TGIDs of tasks in the group;
- `cpuset.cpus`, a list of CPUs belonging to the group;

[3]Note that the `cpuset` controller only manages affinities, and is distinct from the `cpu` controller, which manages per-group bandwidth enforcement for non-`SCHED_DEADLINE` tasks, and the `cpuacct` controller, which primarily tracks CPU usage statistics. Per-group throttling is not supported in `SCHED_DEADLINE`.

- and `cpuset.sched_load_balance`, a binary value controlling whether or not tasks should be migrated between the CPUs in the group to more evenly distribute tasks.

Writing 0 to `cpuset.sched_load_balance` generally indicates that child `cpusets` should form their own root domains. For example, consider the following commands.

```
[root ~]# cd /sys/fs/cgroup/cpuset
[root cpuset]# mkdir group0
[root cpuset]# echo 0-3 > group0/cpuset.cpus
[root cpuset]# echo 0 > cpuset.sched_load_balance
[root cpuset]# echo 2318 > group0/cgroup.procs
```

These command create a child `cpuset group0` containing CPUs 0-3. CPUs 0-3 are allocated to their own root domain, with any remaining CPUs being allocated to another root domain. Task 2318 is allocated to `group0`, which will set task 2318 to have affinity for CPUs 0-3.

For version 2, all controllers have a single hierarchy. The root group for all controllers is located at `/sys/fs/cgroup`. The files of interest to us are

- `cgroup.procs` and `cpuset.cpus`, which carry over from version 1;
- `cgroup.subtree_control`, a list of controllers permitted for use in descendent groups;
- `cpuset.cpus.effective`, the subset of `cpuset.cpus` actually allocated to the group (the remaining CPUs in `cpuset.cpus` may be stolen by child groups, as we will describe shortly),
- and `cpuset.cpus.partition`, which signifies whether or not the group has its own root domain.

For example, consider the following commands.

```
[root ~]# cd /sys/fs/cgroup
[root cgroup]# echo +cpuset > cgroup.subtree_control
[root cgroup]# mkdir group0
[root cgroup]# echo 0-3 > group0/cpuset.cpus
[root cgroup]# echo root > group0/cpuset.cpus.partition
[root cgroup]# echo 2318 > group0/cgroup.procs
[root cgroup]# cat group0/cpuset.cpus.effective
0-3
```

```
[root cgroup]# cat cpuset.cpus.effective
4-7
```

These commands are the equivalent of the previously presented version 1 commands. The last two commands, which display `cpuset.cpus.effective` for both `group0` and the root `cgroup`, demonstrate that CPUs 0-3 are in a root domain that is seperate from the root domain of the remaining CPUs 4-7.

Tasks can also be placed into `cgroup`s at thread granularity (rather than thread-group granularity with `cgroup.procs`). In version 1, this is done by writing task PIDs to file `tasks` in the target group's directory. In version 2, this is done by writing 'threaded' to `cgroup.type` and PIDs to `cgroup.threads`.

4.1.4 Priority Inheritance Mutexes

The RT-mutex is a kernel data structure that provides suspension-based mutual exclusion with priority inheritance. The meaning of priority inheritance depends on the scheduling policy of the highest-priority waiter for an RT-mutex. If this policy is a fixed-priority policy (*i.e.*, `SCHED_FIFO` or `SCHED_RR`), then the owner executes as a `SCHED_FIFO` task with the waiter's priority. If this policy is `SCHED_DEADLINE`, the owner executes as a `SCHED_DEADLINE` task with the waiter's period, budget, and relative deadline. Note that unlike traditional priority inheritance, the owner does not execute with the waiter's absolute deadline. This will be clarified when we discuss implementation details later.

The RT-mutex is accessible from userspace via the `futex()` (fast userspace mutex) system call. In simple terms, a futex optimizes for the case that a mutex is uncontended when a task locks or unlocks the mutex. In this case, a task atomically locks or unlocks the futex from within userspace without involving the kernel (hence, fast userspace). If the futex is contended, locking and unlocking are done by calling `futex()`. The desired operation (*e.g.*, lock or unlock) is specified via an operation argument that `futex()` takes. Priority inheritance is requested via this argument. For example, locking with priority inheritance is requested by passing in `FUTEX_LOCK_PI`. For such operations, the task calling `futex()` waits on a RT-mutex corresponding to the futex.

Note that `pthread` mutexes with priority inheritance are built using futexes with priority inheritance, and thus also follow RT-mutex's priority inheritance behavior.

4.1.5 Admission Control

The kernel ACS limits the fraction of the CPUs that real-time tasks (in a Linux context, this refers to tasks of `SCHED_FIFO`, `SCHED_RR`, and `SCHED_DEADLINE`) are permitted to consume. This fraction is configured as `sched_rt_runtime_us` (default 950000) divided by `sched_rt_period_us`, both of which are files in `proc/sys/kernel`. The default values of these files is `sched_rt_runtime_us`/`sched_rt_period_us` = 950000/1000000 = 95%.

This upper limit on the fraction of CPUs is enforced differently for `SCHED_DEADLINE` than `SCHED_FIFO` and `SCHED_RR`. For `SCHED_DEADLINE`, the ACS enforces that the total bandwidth of `SCHED_DEADLINE` tasks is at most the fraction multiplied by the total CPU capacity. *Capacity* can be thought of as the kernel term for speed (with a maximum of 1.0) under UNIFORM. Restated using the notation established in Chapter 2, the ACS enforces

$$\sum_{\tau_i \in \tau_{\text{act}}(t)} u_i \leq \frac{\texttt{sched\_rt\_runtime\_us}}{\texttt{sched\_rt\_period\_us}} \cdot \sum_{\pi_j \in \pi} sp^{(j)} \tag{4.1}$$

for `SCHED_DEADLINE` tasks. Note that the total bandwidth $\left(\sum_{\tau_i \in \tau} u_i\right)$ and total capacity $\left(\sum_{\pi_j \in \pi} sp^{(j)}\right)$ are computed separately for each root domain.

The ACS will reject the addition of tasks to `SCHED_DEADLINE` (via `sched_setattr()` or `chrt`) if doing so would violate (4.1). The ACS can also reject changes to affinities. Calls to function `sched_setaffinity()` will fail if the requested affinity is not a superset of the root domain (*i.e.*, a `SCHED_DEADLINE` task must be runnable on every CPU in its root domain). To the author's knowledge, a `SCHED_DEADLINE` task will not execute on CPUs outside of its root domain, even if said task has affinity for those CPUs. Likewise, modifying root domains by writing to `cpuset.cpus` in a group with `SCHED_DEADLINE` tasks will fail if the change in CPUs would violate (4.1).

Besides enforcing (4.1) for `SCHED_DEADLINE` tasks, the ACS also tracks the total runtime of real-time tasks (including `SCHED_DEADLINE`). When this total runtime exceeds `sched_rt_runtime_us` multiplied by the number of CPUs, `SCHED_FIFO` and `SCHED_RR` tasks are prevented from executing (note

that SCHED_DEADLINE tasks contribute to, but are not limited by, the tracked total runtime). This tracked total runtime is reset approximately every sched_rt_period_us.[4]

The ACS can be disabled by writing -1 to sched_rt_runtime_us.

```
[root ~]# cd /proc/sys/kernel
[root kernel]# echo -1 > sched_rt_runtime_us
```

This disables bandwidth management for SCHED_FIFO and SCHED_RR. SCHED_DEADLINE tasks are still throttled based on their per-task budget and period, but the limit on total bandwidth is disabled. Restrictions on affinities for SCHED_DEADLINE tasks are also disabled.

4.1.6 Dynamic Voltage and Frequency Scaling

Dynamic Voltage and Frequency Scaling (DVFS) is controlled via the CPUFreq subsystem mounted at /sys/devices/system/cpu/cpufreq. Within this directory is a folder for each *CPUFreq policy* (*e.g.*, policy0, policy1, *etc.*). A CPUFreq policy is a set of CPUs that share DVFS settings due to the underlying hardware. The files of interest within such a CPUFreq policy folder are:

- affected_cpus, the list of CPUs belonging to the CPUFreq policy;
- scaling_governor, the current *governor*, *i.e.*, frequency selection algorithm, managing the CPUs in this policy;
- and scaling_available_governors, the list of compatible governors for this policy.

A user changes the governor of a policy by writing one of the governors in scaling_available_governors to scaling_governor. Governors may not be present in scaling_available_governors depending on the drivers in use in the kernel.

Of relevance to SCHED_DEADLINE is the schedutil governor, which makes frequency scaling decisions based on the real-time parameters of SCHED_DEADLINE tasks. A goal of schedutil is to safely reduce frequencies without incurring deadline misses. The implementation of schedutil will be discussed in Section 4.4.9.

[4]We have purposely oversimplified this description of enforcement of SCHED_FIFO and SCHED_RR because it does not affect SCHED_DEADLINE, which is our focus. In the actual implementation, the tracked total runtime is tracked on a per-CPU basis. A runtime balancing function redistributes this per-CPU runtime when one CPU exceeds sched_rt_runtime_us before the others. This is further complicated by cgroup RT-group scheduling, which permits each group to specify its own sched_rt_runtime_us and sched_rt_period_us. Note that group scheduling is not implemented for SCHED_DEADLINE.

4.2 Common Data Structures

We begin detailing the implementation by describing common data structures (*i.e.*, those not specific to particular Linux policies such as `SCHED_DEADLINE`).

Affinity mask types. The kernel contains many data types representing affinity masks, which we cover in this paragraph. Affinity masks are stored in the kernel as bitmasks. The fundamental bitmask data type is `unsigned long[]`. An array typing is necessary because the maximum number of CPUs possible in the system is configurable. The length of the array is set such that the total number of bits is at least the maximum number of CPUs. Type `struct cpumask` is this array packaged into a `struct`. Type `cpumask_t` is a `typedef` of `struct cpumask`. The last affinity mask type is `cpumask_var_t`. `cpumask_var_t` behaves like a pointer to a `struct cpumask`. The kernel can be configured such that the memory pointed to by a `cpumask_var_t` is allocated either dynamically or on the stack. The exact definition of `cpumask_var_t` changes depending on this configuration. Affinity masks are manipulated via several kernel functions and macros. For example, `cpumask_and()`, which takes three `cpumask_var_t`s (or pointers to either `struct cpumask` or `cpumask_t`), writes into the first mask the bitwise-and of the other two masks.

Red-black trees. The Linux kernel's implementation for balanced binary trees is a red-black tree whose nodes are of type `struct rb_node`. Each `rb_node` contains, as members, pointers to its two children and parent `rb_node`s. A tree as a whole is referenced by a `struct rb_root`, which contains a pointer to the root `rb_node`.

Some priority queues in the scheduler are implemented using red-black trees. These priority queues are the basis of task runqueues, detailed later. In these priority queues, `rb_node`s are ordered in the tree by priority. A pointer to the leftmost `rb_node` is kept, as this `rb_node` corresponds to the highest priority (*e.g.*, for `SCHED_DEADLINE`, `rb_node`s are ordered by deadline, with the smallest, *i.e.*, earliest, deadline having highest priority). The tree is referenced by a `struct rb_root_cached`, which contains an `rb_root` and the aforementioned pointer to the leftmost `rb_node`.

▼ **Example 4.1.** Figure 4.1 illustrates the structure of a red-black tree in Linux. The data structure being stored on this tree are of type `struct sched_dl_entity`, which describes a `SCHED_DEADLINE` task (to be discussed in Section 4.4.1). The illustrated tree orders the `sched_dl_entity`s by their

deadline parameters[5] (*e.g.*, the sched_dl_entity with deadline of 1, the earliest deadline value, is leftmost in the tree). Observe that the rb_root_cached contains pointers to both the root and leftmost nodes. ▲

High-resolution timers. A high-resolution timer (struct hrtimer) is one of the kernel's mechanism for executing code at a specified future time. An example of use in SCHED_DEADLINE is budget enforcement, which we will detail later. An hrtimer contains a member function(), which is a callback function pointer to be called when the hrtimer expires. function() takes a pointer to its corresponding hrtimer as its *only* argument.

A plethora of kernel functions exist for arming and manipulating hrtimers. An hrtimer is initialized by calling hrtimer_init(). An hrtimer is started by calling hrtimer_start(). A started hrtimer can be canceled with hrtimer_try_to_cancel() (which fails if function() is mid-execution) or hrtimer_cancel() (which blocks while function() is mid-execution). An hrtimer can also restart itself from within function(). function() indicates that its corresponding hrtimer should be restarted by returning HRTIMER_RESTART; otherwise, function() should return HRTIMER_NORESTART. These return values are of type enum hrtimer_restart. The firing time of a restarted hrtimer can be set by calling hrtimer_forward() from within function().

Tasks and classes. In Linux, tasks are represented by data structures of type struct task_struct (to be defined in Listing 4.1). These task_structs are partitioned between five *scheduling classes*. These classes exist in a hierarchy such that a task_struct in a given class may only be scheduled if no task_struct in a higher-priority class is schedulable. stop_sched_class, the highest-priority class, and idle_sched_class, the lowest-priority class, are used exclusively by the kernel. By this, we mean that no user-level programs ever belong to these classes.

In order of highest to lowest priority, the remaining classes that govern user-level programs are dl_sched_class (implements SCHED_DEADLINE), rt_sched_class (SCHED_FIFO and SCHED_RR), and the default fair_sched_class (SCHED_NORMAL, SCHED_BATCH, and SCHED_IDLE).

As mentioned in parentheses, each of these three classes corresponds to at least one policy. The static differences in priority between policies results from the hierarchy of scheduling classes. For classes with

[5]To reflect that task parameters are stored as 64-bit integers in SCHED_DEADLINE, we omit trailing zeros (*e.g.*, '1' vs. '1.0') from examples related to SCHED_DEADLINE.

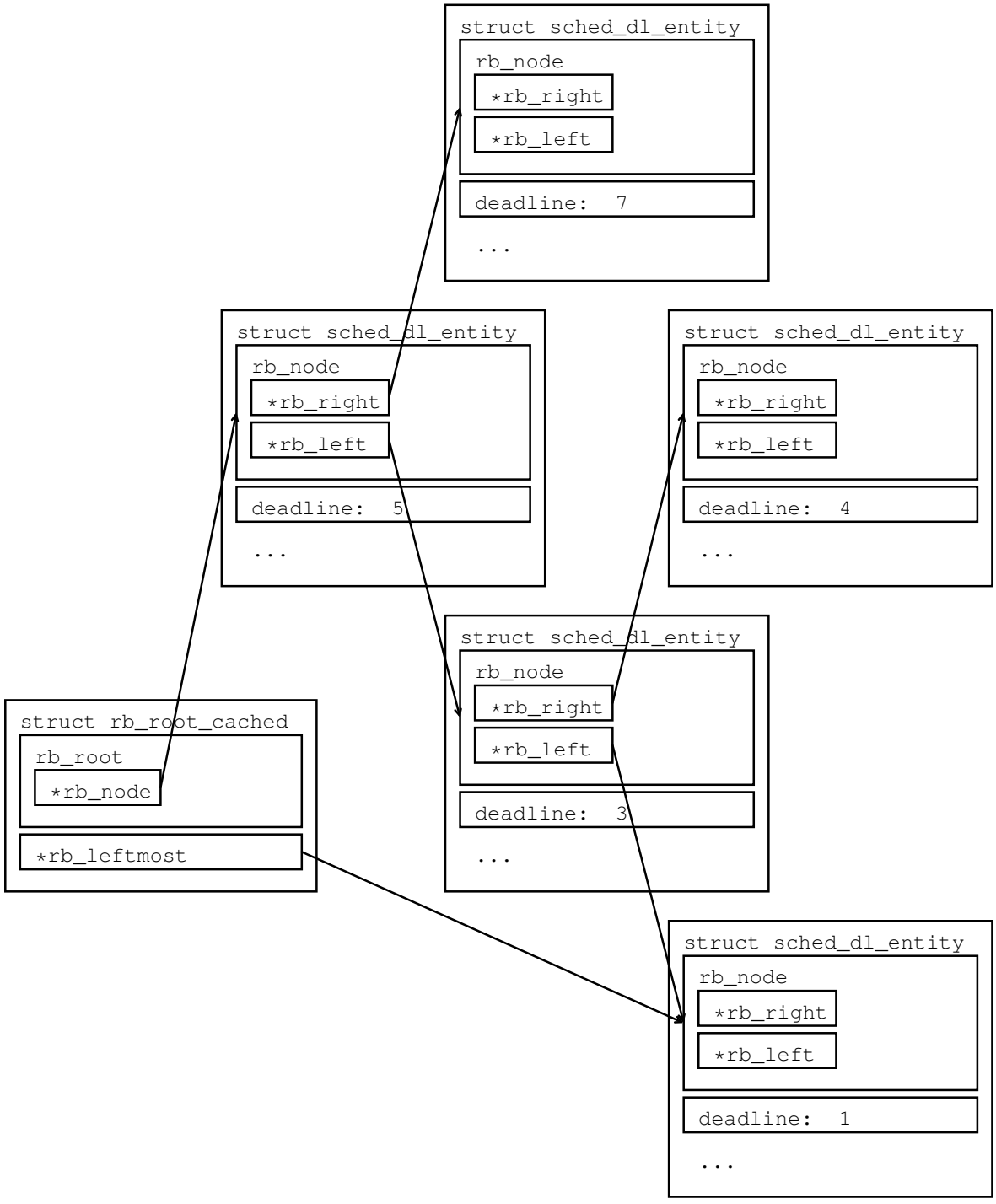


Figure 4.1: `rb_root_cached` example.

```
struct task_struct {
    unsigned int            __state;

    int                     on_rq;

    int                     prio;
    int                     static_prio;
    int                     normal_prio;
    unsigned int            rt_priority;

    struct sched_entity     se;
    struct sched_rt_entity  rt;
    struct sched_dl_entity  dl;

    struct sched_class      *sched_class;
    unsigned int            policy;
    int                     nr_cpus_allowed;
    cpumask_t               *cpus_ptr;
    cpumask_t               cpus_mask;
    unsigned short          migration_disabled;
    unsigned short          migration_flags;

    struct rb_node          pushable_dl_tasks;
};
```

Listing 4.1: `struct task_struct`.

multiple policies, the specific policy chosen with `sched_setattr()` generally causes minor changes in how its corresponding class behaves. For example, `rt_sched_class` supports the policies `SCHED_FIFO`, which tiebreaks equal-priority tasks in FIFO order, and `SCHED_RR`, which executes equal-priority tasks in a time-sliced round-robin fashion.

Scheduling entities. Each of the three user-level scheduling classes defines its own *scheduling entity* data type that stores per-task parameters used by the corresponding scheduler. Each `task_struct` contains a scheduling entity for each of the three user-level scheduling classes.

For example, the scheduling entity type for `SCHED_DEADLINE` is `struct sched_dl_entity`. In `SCHED_DEADLINE`, each thread runs in its own CBS, whose parameters are stored in the corresponding `sched_dl_entity`. These include static CBS parameters such as `dl_runtime` (maximum budget), `dl_deadline` (relative deadline), `dl_period`, and dynamic CBS parameters `runtime` (current budget) and `deadline` (current deadline).

Having covered task_structs, scheduling classes, and scheduling entities at a high level, we can now discuss specific members of these data structures. A subset of the members of task_struct is shown in Listing 4.1.

__state stores the current runnable state of the corresponding task_struct. Suspending and waking functions modify __state to inform the scheduler whether the task_struct can be scheduled. A value of TASK_RUNNING (a macro for 0) indicates that the task_struct is runnable, while nonzero values (*e.g.*, TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBLE, TASK_WAKING, TASK_DEAD, *etc.*) indicate otherwise. A suspended task usually has state TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE. A task in the process of being woken from suspension has state TASK_WAKING. A task that exits has state TASK_DEAD.

on_rq stores whether the corresponding task_struct is on a runqueue or not. on_rq will be discussed after discussing per-CPU runqueues.

Next are the per-task priority values prio, static_prio, normal_prio, and rt_priority. static_prio and rt_priority are scheduling class-specific priority values for fair_sched_class and rt_sched_class, respectively. static_prio stores the nice value (with the addition of a constant offset) for EEVDF tasks. rt_priority stores the priorities of fixed-priority tasks. prio and normal_prio, on the other hand, apply for tasks of all classes. These values lie in the range of $[-1, 140)$. normal_prio is such that dl_sched_class tasks (SCHED_DEADLINE) have value -1, rt_sched_class tasks range between $[0, 100)$, and fair_sched_class between $[100, 140)$. Constants MAX_DL_PRIO (0) and MAX_RT_PRIO (100) are defined such that SCHED_DEADLINE tasks have normal_prio $<$ MAX_DL_PRIO and SCHED_FIFO and SCHED_RR tasks have MAX_DL_PRIO $\leq$ normal_prio $<$ MAX_RT_PRIO. prio usually is equal to normal_prio, but changes to reflect priority inheritance.

se, rt, and dl are the scheduling entities for fair_sched_class, rt_sched_class, and dl_sched_class, respectively. sched_class points to the current scheduling class of the task. Note that sched_class may change due to priority inheritance. policy reflects the scheduling policy chosen with sched_setscheduler(). Note that policy is not modified by priority inheritance.

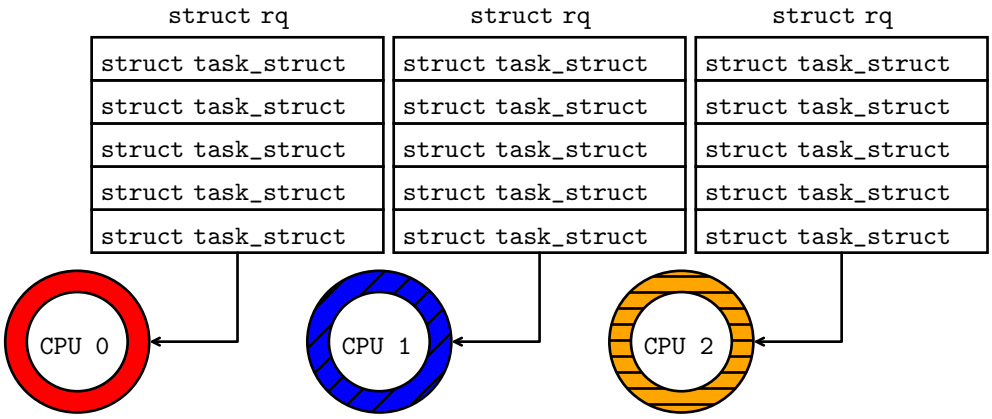


Figure 4.2: Per-CPU runqueues.

`cpus_mask` is the affinity bitmask set by `sched_setaffinity()`.[6] `nr_cpus_allowed` is the number of CPUs in `cpus_mask`. `cpus_ptr` is a pointer to the current affinity bitmask of the corresponding task. `cpus_ptr` usually points to `cpus_mask`, but may point elsewhere when migration is disabled.

Migration can be disabled in the kernel with function `migrate_disable()`. Each call to `migrate_disable()` is paired with a call to `migrate_enable()`. Calls may be nested. Member `migration_disabled` is a counter tracking the number of `migrate_disable()` calls that have yet to be paired with a call to `migrate_enable()`. `migration_flags` is a set of flags that, at time of writing, can only contain `MDF_PUSH`. Usage of `migration_flags` will be described in Section 4.3.8.

Per-CPU runqueues. In Linux, each CPU corresponds to runqueue defined by `struct rq` (see Figure 4.2). The `task_struct` at the head of a CPU's runqueue `rq` is scheduled on said CPU. Internally, each of these `rq`s is composed of sub-runqueues corresponding to the user-level scheduling classes (see Figure 4.3). Not shown in Figure 4.3 is that each sub-runqueue in a `rq` has unique internal structure. For example, the `dl_sched_class` sub-runqueue structure is `struct dl_rq`. Within a `dl_rq`, tasks are ordered by deadline using a red-black tree. On a reschedule for a CPU, the sub-runqueues in the corresponding `rq` are checked in

[6]This is a simplification. The bitmask set by `sched_setaffinity()` is actually pointed to by `user_cpus_ptr`, a different member of `task_struct`. The use case for keeping `cpus_mask` and `user_cpus_ptr` distinct is for heterogeneous architectures such that certain binaries can only run on a subset of CPUs (see the documentation on asymmetric 32-bit execution by Deacon (2021)). If a task changes to a different binary via an `execve()` system call, the new binary may be unable to run on all the CPUs in the requested affinity bitmask. This makes it necessary to set `cpus_mask` to the subset of CPUs allowed by the hardware. The original affinity mask is pointed to by `user_cpus_ptr` such that `cpus_mask` can be restored in the event the task changes back to a binary without CPU restrictions. We choose to avoid discussing `user_cpus_ptr` because `SCHED_DEADLINE` tasks are not expected to call `execve()`.

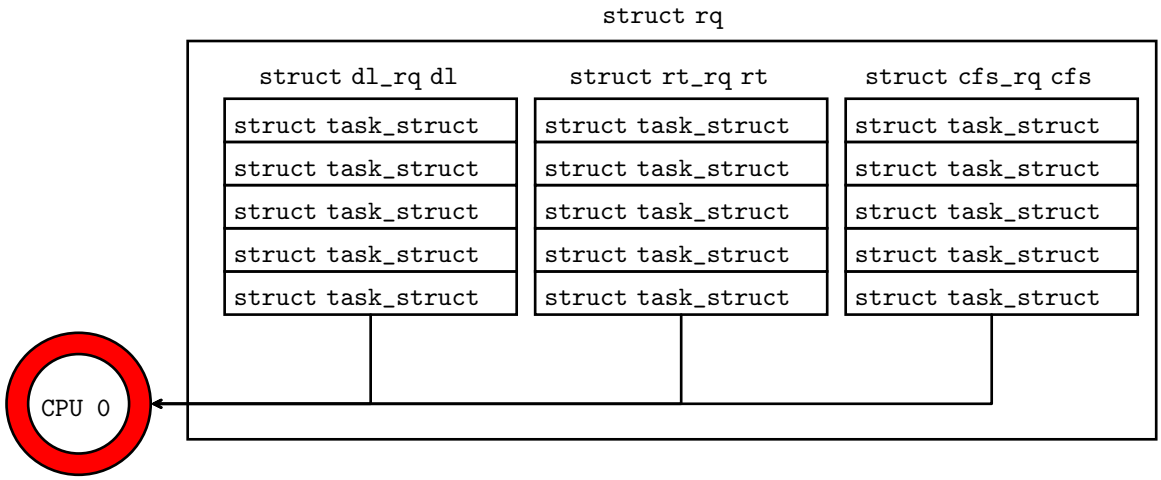


Figure 4.3: Runqueue `rq` is constructed of sub-runqueues.

a consistent order. Thus, the `task_struct` at the head of the first non-empty sub-runqueue is at the head of the `rq`, and is scheduled. The order of the sub-runqueues results in the hierarchy between the scheduling classes.

`cfs`, `rt`, and `dl` are the sub-runqueues corresponding to `fair_sched_class`, `rt_sched_class`, and `dl_sched_class`, respectively. Note their order as members within `struct rq` reflects their order of addition to the kernel, and not the order they are checked on reschedule.

`curr` is a pointer to the currently scheduled task on the `rq`'s CPU.

`idle` and `stop` are pointers to per-CPU kernel-only tasks belonging to `idle_sched_class` and `stop_sched_class`, respectively. Observe in Figure 4.3 and Listing 4.2 that neither of these classes has a corresponding sub-runqueue. Sub-runqueues are unnecessary because `idle` and `stop` are the only tasks of their respective classes that may execute on their corresponding CPU. `stop`, belonging to `stop_sched_class`, has highest priority on `rq` whenever `stop` is runnable. The purpose of `stop` and `stop_sched_class` will be detailed later in Section 4.3.8. `idle`, belonging to `idle_sched_class`, has lowest priority on `rq`, and runs only when no other task is available. The function of `idle` is to transition the CPU into a low-power state. Note that, barring the CPU being offline, `idle` is always runnable.

`clock` stores the most recently read value from the corresponding CPU's clock. `clock_task` and `clock_pelt` are derived from `clock`, and will be discussed in Sections 4.4.3 and 4.4.9. These clock values are accessed by `rq_clock()`, `rq_clock_task()`, and `rq_clock_pelt()`. The clock values are

```
struct rq {
    struct cfs_rq           cfs;
    struct rt_rq            rt;
    struct dl_rq            dl;

    struct task_struct      *curr;

    struct task_struct      *idle;
    struct task_struct      *stop;

    u64                         clock;
    u64                         clock_task;
    u64                         clock_pelt;

    struct root_domain      *rd;
    struct sched_domain     *sd;

    struct balance_callback *balance_callback;

    struct hrtimer          hrtick_timer;

    int                     cpu;

    unsigned int            push_busy;
    struct cpu_stop_work    push_work;
}
```

Listing 4.2: `struct rq`.

updated by calling `update_rq_clock()` on the corresponding `rq`. We omit these calls in our presented pseudocode for simplicity. These clock values are used for tracking budget consumption and in DVFS.

`rd` is a pointer to the unique root domain (recall from Section 4.1.3 that a root domain is similar to a cluster in real-time terminology) that contains the corresponding CPU. The scheduler will only migrate tasks on a runqueue in `rd` to other runqueues in `rd`. `sd` is a pointer to a *scheduling domain*. Like a root domain, a scheduling domain represents a set of CPUs. Unlike root domains, scheduling domains exist in a hierarchy. For two scheduling domains in the same hierarchy, the domain lower in the hierarchy spans a subset of the CPUs in the higher domain. Root and scheduling domains will be detailed in later paragraphs in this section.

`balance_callback` allows its owning runqueue to defer migrations involving this runqueue to a later point in time. `balance_callback` is a linked-list-based queue of callback functions taking a pointer to the corresponding runqueue `rq` as an argument. The functions queued onto `balance_callback` perform the deferred migrations, and are triggered at specific points in the scheduling code (*e.g.*, after any reschedule on a CPU). Deferring migrations with `balance_callback` is useful because critical sections that modify runqueue and task state must be protected by spinlocks and migrations may need to drop these spinlocks. It is necessary to delay such migrations until after such critical sections have completed.

For example, suppose a `SCHED_DEADLINE` task is preempted on a CPU when another task replenishes its budget on that CPU. Processing the preemption requires owning this CPU's runqueue's spinlock. Suppose further that the preempted task has a sufficiently early deadline to preempt the running task on another CPU. Migrating the task between the two runqueues requires holding both runqueues' spinlocks. To avoid deadlock, these per-runqueue spinlocks must be acquired in CPU-index order. If the original CPU has a higher index, it must drop its own spinlock before acquiring the other runqueue's spinlock. Because the original CPU's runqueue lock cannot be dropped while processing the preemption, a function to perform the migration is queued onto `balance_callback`. This function will be triggered by the scheduler after the preemption is processed.

`hrtick_timer` is an `hrtimer` for budget enforcement. Because a task must be executing on a CPU to consume budget, `hrtick_timer` is stored in the CPU's runqueue `rq` rather than in the consuming task's `task_struct`.

`cpu` is the CPU index corresponding with the runqueue.

push_busy is a boolean that indicates whether push_work is in use by the scheduler. push_work is a data structure of type struct cpu_stop_work used by stop_sched_class. Usage of push_work will be described in Section 4.3.8.

Additional task_struct members. Member on_rq of task_struct was not previously described. on_rq has possible values 0, indicating the corresponding task is not on any runqueue (*i.e.*, it is suspended or dead), TASK_ON_RQ_QUEUED, indicating the task is enqueued on some runqueue, and TASK_ON_RQ_MIGRATING, indicating the task is in a transient period in which it is being moved between runqueues.

Each task_struct also stores the CPU index of the last runqueue it was on. How exactly this CPU index is stored depends on the kernel configuration (thus, it was omitted from Listing 4.1). Functions task_cpu() and __set_task_cpu() are defined to get and set this field regardless of configuration. For a given task_struct pointer p, the CPU is returned with task_cpu(p) and set with __set_task_cpu(p, new_cpu). Note that task_cpu() is only guaranteed to be valid if p is queued on a runqueue. This must be determined by observing p->on_rq.

task_rq() is the composition of cpu_rq() and task_cpu(). As such, for task_struct p, task_rq(p) returns a pointer to p's runqueue. Keep in mind that it is not guaranteed that task p is enqueued on this runqueue when task_rq() is called.

pushable_dl_tasks is a red-black tree node. SCHED_DEADLINE, to make migrations more efficient, maintains a deadline-ordered red-black tree of the subset of migratable (*i.e.*, nr_cpus_allowed > 1) SCHED_DEADLINE tasks enqueued on each runqueue. pushable_dl_tasks is the node used to insert the containing task_struct into this tree. Note that this tree is distinct from the tree of all enqueued SCHED_DEADLINE tasks on a runqueue. For this latter tree, the node used to insert the task is stored within sched_dl_entity dl.

Root and scheduling domains. As described in Section 4.1.3, root domains are synonymous with clusters in real-time clustered scheduling. Tasks executing on CPUs in a root domain do not execute on outside CPUs. Root domains are represented by struct root_domain in the kernel. The subset of SCHED_DEADLINE-related members of root_domain are presented in Listing 4.3. span points to the CPU bitmask representing the CPUs in the corresponding root_domain.

dlo_mask points to the CPU bitmask of CPUs in span whose runqueues are *overloaded* with SCHED_DEADLINE tasks. The term overloaded is inherited form CFS, and generally means that a given CPU's

```
struct root_domain {
    cpumask_var_t       span;

    cpumask_var_t       dlo_mask;
    struct dl_bw        dl_bw;
    struct cpudl        cpudl;
};

struct sched_domain {
    struct sched_domain   *parent;

    unsigned long         span[];
};
```

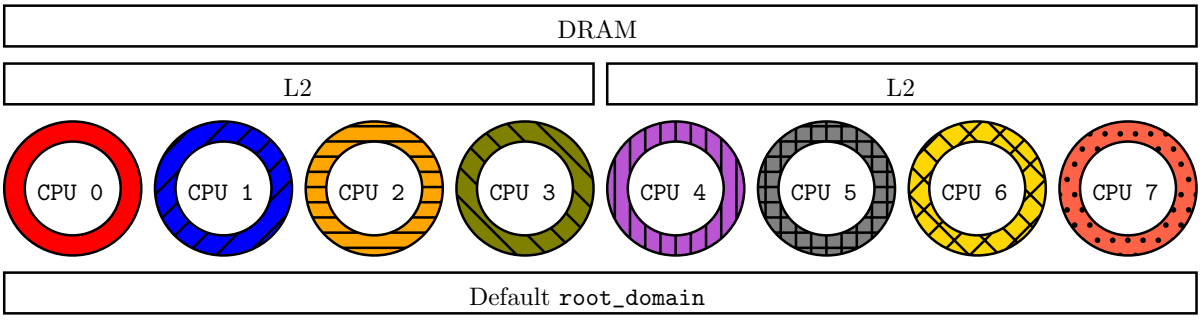
Listing 4.3: `struct root_domain` and `struct sched_domain`.

runqueue has tasks that should be migrated to other CPUs' runqueues. In `SCHED_DEADLINE` specifically, overloaded means that a given CPU's runqueue has at least one unscheduled `SCHED_DEADLINE` task with affinity for another CPU. Note that whether a CPU is overloaded has no relation to the bandwidth of `SCHED_DEADLINE` tasks on said CPU's runqueue. `dlo_mask` is used by `SCHED_DEADLINE` to emulate a global runqueue when rescheduling on a CPU. A global runqueue is emulated by, before rescheduling, migrating higher-priority tasks (than those on the scheduling CPU's runqueue) from other CPUs' runqueues to the scheduling CPU's runqueue. This migration is called a *pull*, to be detailed later in Section 4.4.2. Pulling is made more efficient by only checking the runqueues of CPUs whose corresponding bit is set in `dlo_mask`, as only these runqueues will have migratable `SCHED_DEADLINE` tasks.

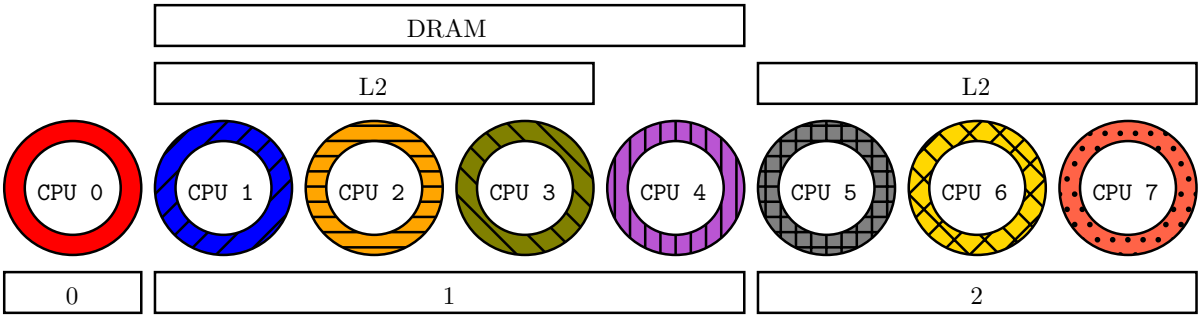
`dl_bw` is a data structure of type `struct dl_bw` whose primary purpose is to track the total bandwidth of `SCHED_DEADLINE` tasks executing on the CPUs in the `root_domain`. This is used for admission control. The members of `struct dl_bw` will be detailed later in Section 4.4.1.

`cpudl` is a data structure of type `struct cpudl`. `cpudl` is primarily a deadline-ordered max-heap of the CPUs in `span`. This heap is used by the scheduler to find the CPU executing the task with latest deadline. A `SCHED_DEADLINE` task that wakes or replenishes its budget may need to be migrated from its current CPU to this latest-deadline CPU. This migration is called a *push*. Pushes and `struct cpudl` will be discussed later in Section 4.4.2.

The CPUs in a `root_domain` are further organized into a hierachy of `sched_domain`s. A `sched_domain` is a set of CPUs within a `root_domain` that are related by level of shared memory (*e.g.*, logical CPUs on the same core under simultaneous multithreading (SMT), CPUs sharing a cache, *etc.*). `sched_`



(a) Example sched_domains with default root_domain.



(b) Example sched_domains with separate root_domains.

Figure 4.4: sched_domain and root_domain illustrations for Example 4.2.

domains indicate to the scheduler sets of CPUs between which migrations are efficient (*e.g.*, migrating between CPUs that share a cache may avoid cache misses). The hierarchy of sched_domains reflects the memory hierarchy, with migrations between CPUs of sched_domains lower in the hierarchy being preferable to sched_domains higher in the hierarchy.

▼ **Example 4.2.** Consider a platform with eight CPUs such that CPUs 0-3 and 4-7 share L2 caches. Suppose the only root_domain is the default root_domain. Figure 4.4a shows that all CPUs fall under the default root_domain (illustrated below the CPUs). CPUs 0-3 and 4-7 also fall under two separate sched_domains due to these two sets of CPUs sharing L2 caches. Both of these sched_domains fall under a parent sched_domain due to all CPUs sharing the same DRAM memory.

Now suppose the system is configured such that three root_domains are created with CPU 0 in root_domain 0, CPUs 1-4 in root_domain 1, and CPUs 5-7 in root_domain 2. These root_domains are illustrated in Figure 4.4b. The sched_domains of the system are modified as a result of creating these root_domains. CPU 0 is not contained in any sched_domain because tasks do not migrate to or away from CPU 0 due to root_domain 0 containing no other CPUs (the kernel

does not create a sched_domain for a single CPU). CPUs 1-3 fall under a sched_domain due to sharing an L2 cache. CPU 4, also in root_domain 1, does not fall under this sched_domain, but does share a higher sched_domain with CPUs 1-3. CPUs 5-7 fall under a sched_domain due to sharing an L2 cache. Because this sched_domain already contains all CPUs in root_domain 2, a higher sched_domain is not created. ▲

sched_domains are primarily used by EEVDF for balancing. The only members relevant to SCHED_DEADLINE are parent, a pointer to the next sched_domain in the hierarchy (or NULL if no such sched_domain exists), and span, the CPU bitmask spanning the sched_domain.

4.3 Scheduling Class Internals

The Linux scheduler contains a common infrastructure that is shared by all scheduling classes. This common infrastructure contains functions for scheduler operations (*e.g.*, rescheduling, suspending and waking tasks, modifying task properties such as policy or CPU affinity) that internally call scheduling-class-specific functions. For example, suspending a SCHED_DEADLINE task will cause the common infrastructure to dequeue the corresponding task_struct from its runqueue rq. The common infrastructure then calls the SCHED_DEADLINE-specific dequeue_task() function to remove task_struct from the dl_rq in said rq.

Each of stop_sched_class, dl_sched_class, rt_sched_class, *etc.*, are a function pointer table (of type struct sched_class) of these class-specific functions. Sections 4.3.1-4.3.8 present pseudocode for the scheduler operations discussed in the previous paragraph. This is done to illustrate where and when class-specific functions are called by the common infrastructure. Note that a sched_class need not provide every function in struct sched_class. Thus, only the functions implemented in dl_sched_class will be discussed.[7]

[7]This omits yield_to_task(), task_dead(), and task_change_group(), which are unique to fair_sched_class; get_rr_interval(), which is used by fair_sched_class and rt_sched_class; and task_fork(), which does nothing in SCHED_DEADLINE.

4.3.1 Scheduling and Suspending

__schedule() (presented in Listing 4.4[8]) is the kernel scheduling function. __schedule() can get called for several reasons including a task becoming unrunnable (by exhausting its budget or suspending) or becoming runnable (by replenishing its budget or waking).

Initially, the CPU calling __schedule() is returned by calling smp_processor_id(). The runqueue rq of said CPU is returned by calling cpu_rq(). The previously scheduled task is set by referencing curr.

At line 9, rq->hrtick_timer is canceled if armed. Why this is necessary will be discussed in Section 4.3.3.

At line 11, prev->__state is checked for a nonzero value. A nonzero value here indicates that __schedule() has been called from some suspension system call (*e.g.*, nanosleep()) that modified __state to a value other than TASK_RUNNING (0). If __state indicates that prev was suspended, prev needs to be removed from runqueue rq so that it cannot be selected for scheduling. __schedule() unsets prev->on_rq and calls the class-specific dequeue_task() in prev->sched_class.

The prototype of dequeue_task() is as follows.

```
void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
```

Besides the runqueue being dequeued from rq and task being dequeued p, dequeue_task() also takes a flags argument that indicates the reason p is being dequeued. The flags relevant to SCHED_DEADLINE are DEQUEUE_SLEEP and DEQUEUE_SAVE. DEQUEUE_SLEEP indicates that p is being dequeued due to a suspension. DEQUEUE_SAVE indicates that p is being dequeued in preparation for modifying p (*e.g.*, its scheduling class or affinity). Such calls to dequeue_task() are part of the *change pattern*, detailed in Section 4.3.5.

Starting from line 16, __schedule() performs load balancing. From the perspective of SCHED_DEADLINE, this means emulating a global runqueue by pulling earlier-deadline tasks from other runqueues to rq. The scheduling classes implement their own balancing logic with class-specific function balance().

The prototype of balance() is as follows.

```
int (*balance)(struct rq *rq, struct task_struct *prev, struct rq_flags *rf);
```

[8] __schedule() actually takes an argument sched_mode that we have omitted in Listing 4.4 because it is primarily used for RCU purposes and we have omitted synchronization code.

```
1  void __schedule(void)
2  {
3      struct sched_class *class;
4      struct task_struct *prev, *next;
5      int cpu = smp_processor_id();
6      struct rq *rq = cpu_rq(cpu);
7      prev = rq->curr;
8
9      hrtick_clear(rq);
10
11     if (prev->__state) {
12         prev->on_rq = 0;
13         prev->sched_class->dequeue_task(rq, p, DEQUEUE_SLEEP);
14     }
15
16     for_each_class_range(class, prev->sched_class, &idle_sched_class)
17         if (class->balance(rq, prev))
18             break;
19
20     prev->sched_class->put_prev_task(rq, prev);
21
22     for_each_class(class) {
23         next = class->pick_task(rq);
24         if (next) {
25             class->set_next_task(rq, next, true);
26             break;
27         }
28     }
29
30     rq->curr = next;
31     /* Do actual context switch */
32         ↳ migrate_disable_switch(rq, prev);
33
34     __balance_callbacks(rq);
35 }
```

Listing 4.4: High-level scheduling function.

Note that `rf` is only used by the kernel's locking validator (which we do not discuss), and does not affect scheduling logic. As such, `rf` was omitted in Listing 4.4. `balance()` returns a nonzero value if a task is pulled from another runqueue.

The `for` loop at line 16 only considers `sched_class`es at or lower than `prev->sched_class` in the class hierarchy. A runnable unscheduled task belonging to a higher class should not exist on another runqueue, as such a task should have been pushed to `rq` earlier and preempted `prev`. The `for` loop is exited as soon as any call to `balance()` manages to pull a task, as any task from a lower class that could be pulled would not be scheduled anyway.

Line 20 calls put_prev_task(). The prototype of put_prev_task() is as follows.

```
void (*put_prev_task)(struct rq *rq, struct task_struct *p);
```

put_prev_task() is called on any task_struct p running on rq expected to be unscheduled (though p may actually continue to be scheduled if it remains the highest-priority task on rq). put_prev_task() is used to perform any bookkeeping required by p's sched_class that reflects that p is unscheduled. For example, in Linux, scheduled tasks are not migratable. If p has affinity for multiple CPUs, SCHED_DEADLINE uses put_prev_task() to internally mark that p is now migratable.

Starting from line 22, the next task to schedule is selected. The classes are iterated over in order of the class hierarchy. The highest-priority task in a class is selected by pick_task(). The prototype of pick_task() is as follows.

```
struct task_struct * (*pick_task)(struct rq *rq);
```

pick_task() returns a pointer to the task_struct of the highest priority task belonging to its sched_class or NULL if no task in this sched_class on runqueue rq can run (a task may be on rq but unrunnable if it has no budget).

Once a task is returned with pick_task(), set_next_task() is called. The prototype of set_next_task() is as follows.

```
void (*set_next_task)(struct rq *rq, struct task_struct *p, bool first);
```

set_next_task() is the opposite of put_prev_task() in that it does bookkeeping required when p is about to become the running task on rq. Following the example discussed with put_prev_task(), in SCHED_DEADLINE, set_next_task() marks task p as not migratable due to it being scheduled. Besides being called in __schedule(), both put_prev_task() and set_next_task() can be called from the aforementioned change pattern that modifies tasks. Argument first is used to indicate whether set_next_task() was called from __schedule() (first is true) or from the change pattern (first is false).

Note that the actual code does not directly call pick_task() and set_next_task() in the real code's equivalent of the for loop at line 22. Instead, another sched_class function, pick_next_task(), is called. For all classes except fair_sched_class, all pick_next_task() does is call pick_task() and conditionally call set_next_task() if pick_task() returns a task.

At line 30, __schedule() sets rq->curr and performs the actual context switch from prev to next. Most details about the context switch code are irrelevant to the scheduler logic, but from within this code migrate_disable_switch() is called. Discussion of this function is deferred to Section 4.3.7, which discusses affinities.

The last line of __schedule() in Listing 4.4 calls __balance_callbacks(). __balance_callbacks() triggers the callback functions queued on rq->balance_callback. Frequently, a queued callback function attempts to push prev to another runqueue now that it has been preempted on rq.

4.3.2 Waking

try_to_wake_up() (presented in Listing 4.5) is the kernel function for waking a suspended task. Note that we have omitted some arguments of try_to_wake_up() in this listing. The actual try_to_wake_up() function in the kernel can fail to wake a task depending on p->__state and omitted try_to_wake_up() argument state. Reasons for such wake failures are outside the scope of this description. The actual try_to_wake_up() in the kernel returns an int describing whether the wake up was successful. We have also omitted argument wake_flags from try_to_wake_up(). wake_flags would be some combination of flags (*e.g.*, WF_SYNC, WF_CURRENT_CPU) that would be passed to select_task_rq() (line 8) and wakeup_preempt() (line 21) along with WF_TTWU. Omitting argument wake_flags does not affect SCHED_DEADLINE's logic, as only fair_sched_class considers these flags.

At line 6, the state of the waking task p is set to TASK_WAKING. This is important for a later call to migrate_task_rq() in try_to_wake_up(). The SCHED_DEADLINE version of migrate_task_rq() treats calls from try_to_wake_up() as a special case. SCHED_DEADLINE recognizes that migrate_task_rq() was called from try_to_wake_up() by observing that p->__state is TASK_WAKING.

At line 8, select_task_rq() is called on the task to be woken p. The prototype of select_task_rq() is as follows.

```
int  (*select_task_rq)(struct task_struct *p, int task_cpu, int flags);
```

select_task_rq() allows the corresponding sched_class to select a new runqueue for a task p. select_task_rq() returns a CPU index. For example, in SCHED_DEADLINE, select_task_rq() attempts to select a CPU such that task p has an earlier deadline than any task on said CPU's runqueue. Note

that `select_task_rq()` does not move `p` to the returned CPU's runqueue (this is handled by `enqueue_task()`, called later). `flags` describes the context `select_task_rq()` was called from. `select_task_rq()` is called when a task is woken (`flags` includes `WF_TTWU`, as in Listing 4.5), newly forked (`WF_FORK`), or executes a new binary (`WF_EXEC`). `task_cpu` is a suggested CPU for `select_task_rq()` to return. `select_task_rq()` is usually called with `task_cpu` equal to `task_cpu(p)`. This is because it is likely that `p` ran on this CPU before suspending, meaning the chance is higher that `p` is cache-hot on this CPU.

Returning to line 8, argument `task_cpu` of `select_task_rq()` is `p->wake_cpu`. `wake_cpu` is also set by `__set_task_cpu()`, so `wake_cpu` and `task_cpu(p)` are usually equivalent. There are some edge cases where the two may be distinct. For example, while suspended, task `p` may lose its affinity for CPU `task_cpu(p)`. In such cases, the function modifying `p`'s affinity mask will set `p->wake_cpu` to one of the CPUs `p` has affinity for.

Consider the block beginning at line 11. This block is entered if `task_cpu(p) != cpu`. In words, this means the CPU chosen by `select_task_rq()` is not the CPU whose runqueue task `p` was last enqueued on, *i.e.*, the task is being migrated. `migrate_task_rq()` is called on `p`.

The prototype for `migrate_task_rq()` is as follows.

```
void (*migrate_task_rq)(struct task_struct *p, int new_cpu);
```

`migrate_task_rq()` is always called immediately prior to `task_cpu(p)` changing its value (*i.e.*, before `__set_task_cpu(p)`). Note that `migrate_task_rq()` does not do the work of moving `p` to `new_cpu`'s runqueue. The purpose of `migrate_task_rq()` is to update class-specific statistics used by the scheduling classes. For example, for `SCHED_DEADLINE` bandwidth reclamation (detailed later in Section 4.4.8), `SCHED_DEADLINE` tracks the total bandwidth of tasks on each CPU (more specifically, for CPU `cpu`, the total bandwidth of tasks `p` such that `task_cpu(p) == cpu`). One of the actions of `SCHED_DEADLINE`'s `migrate_task_rq()` function is to subtract the bandwidth of tasks being woken on new CPUs from the total bandwidths of their original runqueues. This will be detailed in Section 4.4.8.

Returning to the block beginning at line 11, in preparation of `p` being enqueued onto `cpu`'s runqueue, `p`'s CPU is set to `cpu` and `ENQUEUE_MIGRATED` is set in enqueue flags `en_flags`.

Task `p` is actually enqueued starting at line 17. The prototype for `enqueue_task()` is as follows.

```
void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
```

```
1  void try_to_wake_up(struct task_struct *p)
2  {
3      int cpu, en_flags = ENQUEUE_WAKEUP;
4      struct rq *rq;
5
6      p->__state = TASK_WAKING;
7
8      cpu = p->sched_class->select_task_rq(p, p->wake_cpu, WF_TTWU);
9      rq = cpu_rq(cpu);
10
11     if (task_cpu(p) != cpu) {
12         p->sched_class->migrate_task_rq(p, cpu);
13         __set_task_cpu(p, cpu);
14         en_flags |= ENQUEUE_MIGRATED;
15     }
16
17     p->sched_class->enqueue_task(rq, p, en_flags);
18     p->on_rq = TASK_ON_RQ_QUEUED;
19
20     if (p->sched_class == rq->curr->sched_class)
21         p->sched_class->wakeup_preempt(rq, p, WF_TTWU);
22     else if (sched_class_above(p->sched_class, rq->curr->sched_class))
23         resched_curr(rq);
24
25     p->__state = TASK_RUNNING;
26     p->sched_class->task_woken(rq, p);
27 }
```

Listing 4.5: High-level waking function.

enqueue_task() enqueues p onto rq. Flags flags provides additional information on the calling context. The flags relevant to SCHED_DEADLINE are ENQUEUE_MIGRATED, which indicates that task_cpu(p) changed while p was dequeued, ENQUEUE_WAKEUP, which indicates enqueue_task() was called from try_to_wake_up(), and ENQUEUE_RESTORE, which pairs with DEQUEUE_SAVE (see the change pattern described in Section 4.3.5), and ENQUEUE_REPLENISH, which indicates that SCHED_DEADLINE should replenish p's CBS budget.

p->on_rq is set to TASK_ON_RQ_QUEUED after the enqueue completes.

Starting from line 20, try_to_wake_up() determines whether newly woken task p should preempt the currently scheduled task on cpu. If the currently scheduled task and p belong to the same class, wakeup_preempt() is called.

The prototype of wakeup_preempt()[9] is as follows.

```
void (*wakeup_preempt)(struct rq *rq, struct task_struct *p, int flags);
```

wakeup_preempt() checks if p has a higher priority than whatever is running on rq. If so, it alerts rq to reschedule by calling resched_curr() on rq. resched_curr() sets a flag indicating to the kernel that the CPU corresponding with rq should call __schedule(). wakeup_preempt() also takes wake flags in argument flags, but this is only used by fair_sched_class, so we do not detail its usage.

If the woken task p is of a higher class than the currently scheduled task, try_to_wake_up() directly calls resched_curr().

try_to_wake_up() indicates that the woken task p is runnable by setting its state to TASK_RUNNING (line 25). task_woken() is also called. The prototype of task_woken() is as follows.

```
void (*task_woken)(struct rq *this_rq, struct task_struct *task);
```

task_woken() is called in try_to_wake_up() and also when task is a newly created task (though newly created tasks cannot be in SCHED_DEADLINE because SCHED_DEADLINE tasks cannot fork). As such, any call to task_woken() comes after an accompanying call to select_task_rq() (with flag WF_TTWU or WF_FORK). The purpose of task_woken() is to double check that task task should be queued on runqueue this_rq that was chosen by the accompanying call to select_task_rq(). The meaning of "should be queued" is class-dependent; for example, in SCHED_DEADLINE, task_woken()

[9]Prior to kernel 6.7, this function was named check_preempt_curr().

```
static enum hrtimer_restart hrtick(struct hrtimer *timer)
{
    struct rq *rq  = container_of(timer, struct rq, hrtick_timer);
    rq->curr->sched_class->task_tick(rq, rq->curr, 1);
                          ↳ rq->curr->sched_class->update_curr(rq);

    return HRTIMER_NORESTART;
}
```

Listing 4.6: HR tick function.

will check that `task` preempts the currently scheduled task on `this_rq`. If `task` should not be queued on `this_rq`, `task_woken()` attempts to migrate `task` to another runqueue.

At first glance, the call to `task_woken()` may seem redundant because we have not made clear why the choice by `select_task_rq()` should be double checked. This is a consequence of omitting synchronization code for simplification. The problem with `select_task_rq()` is that it is called and returns without holding any runqueue spinlocks. The spinlock of runqueue `rq` in Listing 4.5 is only acquired immediately before the call to `enqueue_task()`. Other tasks may have been enqueued onto `rq` in between the calls to `select_task_rq()` and `enqueue_task()` such that `p` is no longer the highest-priority task on `rq` by the time it is enqueued. `task_woken()` differs from `select_task_rq()` because it acquires runqueue spinlocks when deciding where to push tasks to.

4.3.3 Ticks

Ticks are recurring timer interrupts that give the scheduler the opportunity to respond to the current state of the system. For example, on a tick, a scheduler may observe that a running task has exhausted its budget, and thus reschedule on that task's CPU. Ticks are a combination of periodic and event-driven. Periodic ticks occur automatically with some architecture-defined period,[10] typically with on-order-millisecond granularity. Event-driven ticks are based on per-runqueue `hrtimer` `hrtick_timer`, which must be manually armed by the scheduler, but may fire with finer granularity than periodic ticks. We assume a kernel configured with event-driven ticks. Event-driven ticks are executed with `hrtick_timer`'s callback function, `hrtick()` (Listing 4.6).

`hrtick()` is called whenever a runqueue's `hrtick_timer` fires. `hrtick()` is armed and canceled with `hrtick_start()` and `hrtick_clear()`, respectively. These functions are essentially wrap-

[10]These ticks are not strictly periodic for reasons outside the scope of this dissertation.

pers around hrtimer_start() and hrtimer_cancel(). Calling hrtick_start(rq, delay) arms rq->hrtick_timer to fire after delay nanoseconds, and calling hrtick_clear(rq) cancels rq->hrtick_timer. hrtick_start() is typically called within sched_class functions to guarantee that task_tick() is called at some event (*e.g.*, in SCHED_DEADLINE, set_next_task() calls hrtick_start() to fire when the current task's budget would expire such that task_tick() will throttle the task). hrtick_clear() is called at the beginning of __schedule() (recall Listing 4.4). This prevents hrtick() from hitting the wrong task. For example, SCHED_RR relies on periodic ticks to grant timeslices. Suppose a SCHED_DEADLINE task suspends such that a SCHED_RR task is next to be scheduled. If an hrtick() armed by SCHED_DEADLINE were able to fire on this SCHED_RR task, SCHED_RR may incorrectly shorten the task's timeslice.

Consider Listing 4.6. Within hrtick(), a pointer to the runqueue containing hrtick_timer is retrieved with the container_of() macro. container_of() returns a pointer to the struct containing a member given a pointer to the member (*e.g.*, timer in Listing 4.6), the type of the containing struct (*e.g.*, struct rq), and the identifier of the member in the struct (*e.g.*, hrtick_timer). hrtick() then calls sched_class function task_tick() on the runqueue returned by container_of().

The prototype of task_tick() is as follows.

```
void (*task_tick)(struct rq *rq, struct task_struct *p, int queued);
```

task_tick() is called on any scheduler tick (by hrtick() or by periodic tick functions). The main purpose of task_tick() in rt_sched_class and dl_sched_class is to call update_curr(). In fair_sched_class, task_tick() does not call update_curr() directly, but does call a helper function that does much of the work in update_curr(). Argument queued is 1 if task_tick() was called from hrtick() and 0 if it was called from a periodic tick function.

The prototype of update_curr() is as follows.

```
void (*update_curr)(struct rq *rq);
```

update_curr() is mainly called from within task_tick(). There are additional calls to update_curr() from the CPU accounting cgroup controller, but these are not relevant to SCHED_DEADLINE. The purpose of update_curr() is to update scheduling statistics and whatever sched_class-specific statistics are needed by the sched_class for task rq->curr. For example, in theory, server budgets are reasoned about as if they deplete continuously with time. The kernel, being a real system, has to decrement

budgets discretely. In SCHED_DEADLINE, budgets are decremented within update_curr(). When a SCHED_DEADLINE task is scheduled on a CPU, the kernel arms hrtick_timer on said CPU's runqueue to fire when the task's budget would be exhausted. Assuming the task is not preempted, hrtick() is executed at the exhaustion time, which in turn will call task_tick() and update_curr(). This call to update_curr() depletes the task's budget to 0, upon which the CPU should reschedule.

4.3.4 Yielding

Calling sched_yield() from userspace straightforwardly calls class-specific yield_task() in the kernel. The prototype of yield_task() is as follows.

```
void (*yield_task) (struct rq *rq);
```

rq is the runqueue of the CPU executing the calling task of sched_yield(). yield_task() suggests to the scheduler that something else should run. The exact meaning of this varies depending on the specific sched_class.

4.3.5 Change Pattern

Sections 4.3.6-4.3.7 discuss sched_class functions used when changing certain scheduling attributes of tasks. Functions that implement these changes will follow a *change pattern* (Listing 4.7), which we cover in this subsection. For this subsection, the term *change function* denotes a function that uses the change pattern.

The purpose of the change pattern is to place task p into a state where scheduling attributes can be changed without introducing inconsistencies into class-common or -specific data. This means temporarily dequeuing p from any runqueue it might be on and unmarking it as scheduled if necessary. Whether p is on its runqueue and is the current task on its runqueue is stored in booleans queued and running.

Prior to the change function modifying p, dequeue_task() and put_prev_task() are called if necessary. If called, dequeue_task() is called with at least DEQUEUE_SAVE set in queue_flags. DEQUEUE_SAVE indicates that this call came from the change pattern. The change function may set additional flags in queue_flags to provide additional context to dequeue_task() and enqueue_task().

After p's attribute is changed, dequeue_task() and put_prev_task() (if they were called) must be undone with enqueue_task() and set_next_task(). enqueue_task() is also called with

```
1  int queued, running;
2  int queue_flag = DEQUEUE_SAVE; /* Matches ENQUEUE_RESTORE */
3  struct rq *rq = task_rq(p);
4
5  /* Prepare change */
6
7  queued = p->on_rq == TASK_ON_RQ_QUEUED;
8  running = rq->curr == p;
9
10 if (queued)
11     p->sched_class->dequeue_task(rq, p, queue_flag);
12 if (running)
13     p->sched_class->put_prev_task(rq, p);
14
15 /* Change attribute of p */
16
17 if (queued)
18     p->sched_class->enqueue_task(rq, p, queue_flag);
19 if (running)
20     p->sched_class->set_next_task(rq, p, false);
21
22 /* Post-change follow-up */
```

Listing 4.7: Change pattern for `task_struct *p`.

at least `ENQUEUE_RESTORE` set in `queue_flags`. This occurs without additional setting or unsetting in `queue_flags` because flags `DEQUEUE_SAVE` and `ENQUEUE_RESTORE` are defined to be equivalent. `ENQUEUE_RESTORE` serves the same purpose as `DEQUEUE_SAVE`, indicating to `enqueue_task()` that the call originated from the change pattern. `set_next_task()` is called with `false` for the same purpose.

After the change pattern, the change function will usually execute some follow-up code. For example, modifying a task's affinity may cause said task to lose affinity for the CPU whose runqueue it is currently on. Follow-up code will then migrate the task to a CPU it has affinity for.

To simplify the presentation of change functions, in future listings, comments /* *Change pattern start* */ and /* *Change pattern end* */ are shorthand for lines 7-13 and 17-20, respectively.

4.3.6 Policy Changes and Priority Inheritance

Both scheduling policy change requests and priority inheritance can result in changing the scheduler that manages a task. Changing schedulers is implemented by switching a task's `sched_class`, as this swaps

```
int __normal_prio(int policy, int rt_prio, int nice)
{
    int prio;

    if (policy == SCHED_DEADLINE)
        prio = MAX_DL_PRIO - 1;
    else if (policy == SCHED_FIFO || policy == SCHED_RR)
        prio = MAX_RT_PRIO - 1 - rt_prio;
    else
        prio = NICE_TO_PRIO(nice);

    return prio;
}

void __setscheduler_prio(struct task_struct *p, int prio)
{
    if (dl_prio(prio))
        p->sched_class = &dl_sched_class;
    else if (rt_prio(prio))
        p->sched_class = &rt_sched_class;
    else
        p->sched_class = &fair_sched_class;

    p->prio = prio;
}

void check_class_changed(struct rq *rq, struct task_struct *p, struct sched_
   class *prev_class, int oldprio)
{
    if (prev_class != p->sched_class) {
        prev_class->switched_from(rq, p);
        p->sched_class->switched_to(rq, p);
    } else if (oldprio != p->prio || dl_prio(p->prio))
        p->sched_class->prio_changed(rq, p, oldprio);
}
```

Listing 4.8: Class change helper functions.

the class-specific functions called by the common scheduling infrastructure. Before discussing the mechanics of changing a task's `sched_class`, we must first discuss three helper functions (Listing 4.8).

First is `__normal_prio()`, which maps class-specific priority values (*e.g.*, fixed priorities for `rt_sched_class` and nice values for `fair_sched_class`; `dl_sched_class` lacks an analogous static value because EDF priorities are dynamic) into the class-agnostic priority range used by `task_struct` members `prio` and `normal_prio`. `__normal_prio()` is used to compute the value of `normal_prio` upon a change in policy or class-specific priority.

Second is __setscheduler_prio(), which is the function called to change p->sched_class and p->prio for a policy change or priority inheritance. Which sched_class to set p->sched_class to is determined from the new prio value. dl_prio() is true when prio maps into the priority range occupied by dl_sched_class (*i.e.*, -1). rt_prio() is the corresponding function for rt_sched_class.

Last is check_class_changed(), which calls up to three sched_class functions: switched_from(), switched_to(), and prio_changed(). Argument oldprio was the value of p->prio before the caller (a policy change request or priority inheritance function) of check_class_changed() was executed.

The prototype of switched_from() and switched_to() is as follows.

```
void (*switched_from)(struct rq *this_rq, struct task_struct *task);
void (*switched_to)  (struct rq *this_rq, struct task_struct *task);
```

switched_from() and switched_to() are used to perform bookkeeping involved in task changing task->sched_class. For example, in SCHED_DEADLINE, switched_from() reduces the total bandwidth of the root_domain corresponding to this_rq by task's bandwidth. Depending on the corresponding sched_class, switched_from() and switched_to() may also migrate tasks and reschedule this_rq's CPU. For example, in SCHED_DEADLINE, switched_from() may attempt to pull other SCHED_DEADLINE tasks to this_rq. To see why this pulling may be necessary here, suppose task changes its class to fair_sched_class such that task was the scheduled task and the only SCHED_DEADLINE task on this_rq. When this_rq's CPU reschedules, because task now belongs to fair_sched_class, the balance() function for SCHED_DEADLINE will not be called (recall Line 16 of Listing 4.4). Thus, this_rq will not pull SCHED_DEADLINE tasks prior to picking a task to schedule. If there are unscheduled SCHED_DEADLINE tasks on other runqueues, they must be pulled before this reschedule, *i.e.*, during switched_from().

The prototype of prio_changed() is as follows.

```
void (*prio_changed) (struct rq *this_rq, struct task_
    struct *task, int oldprio);
```

prio_changed() is responsible for accounting for changes in the priorities of tasks. For SCHED_DEADLINE, "changes in the priorities" refers to changes due to priority inheritance (and not for standard

deadline changes under the CBS). `prio_changed()` may migrate tasks to or reschedule `this_rq` for similar reasons to `switched_from()` and `switched_to()`.

Note that in `check_class_changed()`, for `SCHED_DEADLINE` tasks, `prio_changed()` is called regardless of whether `oldprio` equals `p->prio`. This is because all `SCHED_DEADLINE` tasks have `prio` equal to `-1`. Thus, whether or not a `SCHED_DEADLINE` task is inheriting priority from another `SCHED_DEADLINE` task cannot be determined by comparing `oldprio` and `prio`.

It remains to discuss how these three helper functions are used by policy change requests and priority inheritance.

Policy changes. As stated in Section 4.1.1, all system calls pertaining to changing scheduling policy are ultimately serviced by kernel function `__sched_setscheduler()`, presented in Listing 4.9.[11] Argument `attr` is a `struct` whose members describe the policy change request. Note that this request may keep the previous policy and only modify a policy's parameters. For example, `__sched_setscheduler()` may be called to modify a `SCHED_DEADLINE` task's period. `__sched_setscheduler()` returns 0 if the requested change is successful.

`__sched_setscheduler()` must verify that the change request `attr` is permitted by the ACS. For example, the ACS may reject `attr` if it converts `p` to a `SCHED_DEADLINE` task whose bandwidth would overload `p`'s runqueue's `root_domain`. The exact checks performed on `attr` will be described in Section 4.4.4. The ACS rejecting the request will cause `__sched_setscheduler()` to fail, returning some error number depending on the cause of the rejection.

If admission control is passed, `p->normal_prio` is set by calling `__normal_prio()` (line 13) with the members of `attr` as arguments. The resulting priority value is also stored in `newprio`, the tentative new value of `p->prio`.

At line 15, `rt_mutex_get_top_task()` is called. `rt_mutex_get_top_task()` returns a pointer to the highest-priority `task_struct` waiting for an RT-mutex owned by `p`, or `NULL` if no such task exists. If a task is returned, it is written to `pi_task`, and `newprio` is set as the higher priority value between the current value of `newprio` (recall that this is the priority value not affected by priority inheritance, `p->normal_prio`) and `pi_task->prio`.

[11] Note that the actual `__sched_setscheduler()` function takes two additional boolean arguments, `user` and `pi`. These are omitted in the listing because they are always `true` when `__sched_setscheduler()` is called as a result of a policy change system call.

```
1  int __sched_setscheduler(struct task_struct *p, struct sched_attr *attr)
2  {
3      int policy = attr->sched_policy;
4      int oldprio, newprio, queued, running;
5      int queue_flags = DEQUEUE_SAVE;
6      struct sched_class *prev_class = p->sched_class;
7      struct rq *rq = task_rq(p);
8      struct task_struct *pi_task;
9
10     /* Do admission control */
11
12     oldprio = p->prio;
13     newprio = p->normal_prio = __normal_
           prio(policy, attr->sched_priority, attr->sched_nice);
14
15     pi_task = rt_mutex_get_top_task(p);
16     if (pi_task)
17         newprio = min(newprio, pi_task->prio);
18
19     /* Change pattern start */
20
21     p->policy = policy;
22     __setscheduler_prio(p, newprio);
23     if (policy == SCHED_DEADLINE)
24         __setparam_dl(p, attr);
25
26     /* Change pattern end */
27
28     check_class_changed(rq, p, prev_class, oldprio);
29     rt_mutex_adjust_pi(p);
30
31     return 0;
32 }
```

Listing 4.9: Policy changes.

Once newprio is computed, the request is enacted. p->policy is set to the requested policy. p->sched_class and p->prio are set by calling __setscheduler_prio().

Class-specific parameters are also set (only the setting of dl_sched_class parameters is shown in Listing 4.9). If policy is SCHED_DEADLINE, dl_sched_class parameters are set with function __setparam_dl(). __setparam_dl() sets three sets of parameters in p's sched_dl_entity, p->dl. First are static CBS parameters. These are the budget dl_runtime, relative deadline dl_deadline, and period dl_period, whose values are taken from attr. Second are derived static parameters bandwidth dl_bw, equal to dl_runtime/dl_period, and density dl_density, equal to dl_runtime/dl_deadline. Last is pi_se, a pointer in the p's sched_dl_entity, p->dl. __setparam_dl() sets pi_se to point to its containing sched_dl_entity, p->dl. This is pi_se's default value for when its task is not inheriting another task's priority. pi_se will be detailed later when discussing function rt_mutex_setprio() (Listing 4.10).

After the request is enacted, check_class_changed() is called. check_class_changed() will reschedule and migrate tasks in response to changes in p's priority due to enacting the policy change.

rt_mutex_adjust_pi() is also called to deal with priority inheritance. p's policy change may result in a change in its priority (due to a change in p's sched_class or, in the case of rt_sched_class, a change in p's fixed priority). This priority change needs to be propagated up the chain of RT-mutex owners that is such that it begins with p, it ends with a task not waiting on an owned RT-mutex, and each successive task in the chain owns the RT-mutex that the previous task is waiting on. Note this chain may end with p if it is not a waiter. Conceptually, rt_mutex_adjust_pi() calls rt_mutex_setprio() (discussed in the following paragraph) on each task in this chain to update its inherited priority until a task in the chain has a higher priority than p.

Priority inheritance. Whenever the highest-priority waiter on an RT-mutex changes, function rt_mutex_setprio() (Listing 4.10) is called to implement priority inheritance. Argument p is the owner of the RT-mutex and pi_task is the new highest-priority waiter (or NULL if there are no waiters).

Priority inheritance is conceptually similar to a policy change in that p's sched_class and priority prio are changed. Because most of Listing 4.10 is the same as Listing 4.9, we focus discussion on the lines within the change pattern. We also omit non-SCHED_DEADLINE priority inheritance code. When inheriting from SCHED_DEADLINE tasks, priority inheritance works by setting a pointer in the inheriting task to the

```
1  void rt_mutex_setprio(struct task_struct *p, struct task_struct *pi_task)
2  {
3      int prio, oldprio, queued, running;
4      int queue_flag = DEQUEUE_SAVE;
5      struct sched_class *prev_class = p->sched_class;
6      struct rq *rq = task_rq(p);
7
8      prio = p->normal_prio;
9      if (pi_task)
10         prio = min(pi_task->prio, prio);
11     oldprio = p->prio;
12
13     /* Change pattern start */
14
15     if (dl_prio(prio)) {
16         if (!dl_prio(p->normal_prio) ||
17             (
18                 pi_task &&
19                 dl_prio(pi_task->prio) &&
20                 pi_task->dl.deadline < p->dl.deadline)
21             )
22         ) {
23             p->dl.pi_se = pi_task->dl.pi_se;
24             queue_flag |= ENQUEUE_REPLENISH;
25         } else
26             p->dl.pi_se = &p->dl;
27     }
28
29     __setscheduler_prio(p, prio);
30
31     /* Change pattern end */
32
33     check_class_changed(rq, p, prev_class, oldprio);
34 }
```

Listing 4.10: Priority inheritance.

donor task. More specifically, this priority-inheritance-scheduling-entity pointer, `pi_se`, is stored in and points to the `sched_dl_entity`s of the inheritor and donor tasks.

In `rt_mutex_setprio()`, at line 15, `prio` is the tentative new value for `p->prio` that `p` may be inheriting from task `pi_task`. If `dl_prio(prio)` is `true`, then there are two cases: `p` is either inheriting the priority of a `SCHED_DEADLINE` waiter or `p` is a `SCHED_DEADLINE` task with higher priority than any waiter. The inner `if-else` statements correspond to these two cases.

Consider the condition in lines 16-21. This condition corresponds to `p` inheriting the priority of a `SCHED_DEADLINE` waiter. If `dl_prio(p->normal_prio)` is `false`, then `p` is not a `SCHED_DEADLINE` task. Thus, for `dl_prio(prio)` to have been `true`, `prio` must have been inherited from a

pi_task under SCHED_DEADLINE. Alternatively, if dl_prio(p->normal_prio) is true, then p inherits from pi_task only if pi_task exists (*i.e.*, is not NULL), pi_task is or inherits from a SCHED_DEADLINE task (*i.e.*, dl_prio(pi_task->prio)), and pi_task has an earlier deadline than p (pi_task->dl.deadline < p->dl.deadline).

If the condition in lines 16-21 is true, then p->dl.pi_se is set to pi_task->dl.pi_se to indicate that p is inheriting from pi_task. Note that pi_task->dl.pi_se is used instead of &pi_task->dl because priority inheritance is transitive. ENQUEUE_REPLENISH is also set in queue_flags. This is because SCHED_DEADLINE tasks that inherit priority enter a *boosted* state where budget is replenished immediately on exhaustion (instead of the task being throttled). This boosted state will be described in Section 4.4.7.

If the condition in lines 16-21 is false, then this corresponds with the case that p is a SCHED_DEADLINE task with higher priority than any waiter it could inherit from. pi_se is reset to point to p's sched_dl_entity on line 26 to indicate that p does not inherit another task's priority.

4.3.7 Affinities

Tasks' affinities are set by either the sched_setaffinity() system call, migration enabling/disabling, and the cpuset controller. All of these methods for setting affinities rely on a common set of helper functions shown in Listing 4.11. Each helper function in Listing 4.11 calls the successive helper functions.

The topmost helper function is __set_cpus_allowed_ptr().[12] __set_cpus_allowed_ptr() sets p->cpus_ptr (the true affinity mask, affected by enabling/disabling migration) and possibly sets p->cpus_mask (the backup affinity mask that is unchanged by enabling/disabling migration) depending on flags.

__set_cpus_allowed_ptr() calls __do_set_cpus_allowed(), discussed in the next paragraph, which does the actual work of setting p->cpus_ptr and p->cpus_mask. After the affinity mask is changed in __do_set_cpus_allowed(), p may need to be migrated if affinity for its current CPU is lost. cpumask_any_distribute() returns a CPU index in p's new affinity mask. This CPU is passed

[12]Note that in recent kernels, the arguments (*e.g.*, cpumask pointers and flags) to these helper functions and the class-specific set_cpus_allowed() functions are passed in as members of a struct of type affinity_context instead of individually, as presented in Listing 4.11 and the prototype of set_cpus_allowed(). We are also omitting the user_mask member of affinity_context and SCA_USER flag in our description as these are only used for task_struct member user_cpus_ptr (see Footnote 6).

as a suggested CPU to affine_move_task(), which does the actual work of migrating p if it is queued (if not queued, p will be migrated by try_to_wake_up()). Code for affine_move_task() is not presented due to its complexity. This complexity arises for reasons that are not useful for understanding SCHED_DEADLINE. If affine_move_task() chooses to migrate p, it blocks until p is migrated. Note that affine_move_task() may choose not to migrate p if it did not lose affinity for its current CPU.

affine_move_task() returns 0 once p is on a runqueue corresponding to a CPU it has affinity for. affine_move_task() can also return -EINVAL, but this should only occur if the implementation is somehow buggy.

__do_set_cpus_allowed() uses the change pattern. The affinity change is done by calling the class-specific set_cpus_allowed() function. The prototype for set_cpus_allowed() is as follows.

```
void (*set_cpus_allowed)(struct task_
    struct *p, struct cpumask *newmask, u32 flags);
```

The main function of set_cpus_allowed() is to call set_cpus_allowed_common() (in fact, in fair_sched_class and rt_sched_class, set_cpus_allowed() is set to set_cpus_allowed_common()). SCHED_DEADLINE also uses set_cpus_allowed() to maintain per-root_domain bandwidth totals, which are used by the ACS. If p's change in affinity causes it to change root_domains, then p's bandwidth must be deducted from its original root_domain's bandwidth total. Note that adding p's bandwidth to its new root_domain occurs elsewhere. This will be detailed more in Section 4.4.4. Be aware that no admission control can be performed in set_cpus_allowed(). By the time set_cpus_allowed() is called, the scheduler has already committed to changing p's affinity mask.

set_cpus_allowed_common() is the function that actually sets cpus_ptr and cpus_mask. If the call to set_cpus_allowed_common() originated from a function either enabling or disabling migration, then flags has either SCA_MIGRATE_ENABLE or SCA_MIGRATE_DISABLE set. If so, then the requested change in affinity is transient, *i.e.*, only cpus_ptr should be modified. Otherwise, the call to set_cpus_allowed_common() requested to permanently change p's affinity mask. Then the value of new_mask is written to cpus_mask with cpumask_copy(), and nr_cpus_allowed is updated to reflect the number of CPUs in new_mask with cpumask_weight().

```
1  int __set_cpus_allowed_ptr(struct task_
       struct *p, struct cpumask *new_mask, u32 flags)
2  {
3      unsigned int dest_cpu;
4      struct rq *rq = task_rq(p);
5      struct rq_flags rf;
6
7      __do_set_cpus_allowed(p, new_mask, flags);
8
9      dest_cpu = cpumask_any_distribute(new_mask);
10     return affine_move_task(rq, p, &rf, dest_cpu, flags);
11 }
12
13 void __do_set_cpus_allowed(struct task_
       struct *p, struct cpumask *new_mask, u32 flags)
14 {
15     bool queued, running;
16     struct rq *rq = task_rq()(p);
17
18     /* Change pattern start */
19
20     p->sched_class->set_cpus_allowed(p, new_mask, flags);
21                     ↳ set_cpus_allowed_common(p, new_mask, flags);
22
23     /* Change pattern end */
24 }
25
26 void set_cpus_allowed_common(struct task_
       struct *p, struct cpumask *new_mask, u32 flags)
27 {
28     if (flags & (SCA_MIGRATE_ENABLE | SCA_MIGRATE_DISABLE)) p->`cpus_
          ptr = new_mask;
29         return;
30     }
31
32     cpumask_copy(&p->cpus_mask, new_mask);
33     p->nr_cpus_allowed = cpumask_weight(new_mask);
34 }
```

Listing 4.11: Affinity helpers.

```
int __sched_setaffinity(struct task_struct *p, struct cpumask *mask)
{
    cpumask_var_t cpus_allowed, new_mask;

    cpuset_cpus_allowed(p, cpus_allowed);
    cpumask_and(new_mask, mask, cpus_allowed);

    /* Do admission control */

    return __set_cpus_allowed_ptr(p, new_mask, 0);
}
```

Listing 4.12: Affinity system call.

`sched_setaffinity()`. System call `sched_setaffinity()` calls `__sched_setaffinity()` (Listing 4.12) in the kernel. `cpuset_cpus_allowed()` writes the mask of CPUs permitted by `p`'s `cpuset` to `cpus_allowed`. The reduction of the requested affinity mask `mask` to only the CPUs permitted by this `cpuset` is written to `new_mask`. `new_mask` is then checked by the ACS (to be detailed in Section 4.4.4). If `new_mask` passes admission control, then `__set_cpus_allowed_ptr()` is called to change `p`'s affinity mask to `new_mask`.

Enabling and disabling migration. Disabling migration is a mechanism used in the kernel to protect critical sections that operate on per-CPU data structures. Such critical sections would normally be protected by disabling preemption and/or interrupts on the CPU owning the data structures. This is undesirable for latency because it can prevent the scheduling of high-priority tasks while preemption or interrupts are disabled.

Instead, per-CPU data can be protected by a combination of disabling migration and suspension-based locking. This combination has the advantage that a low-priority task can be preempted mid-critical section. Disabling migration over the critical section guarantees that accesses of per-CPU data reference data belonging to the same CPU. Locking prevents other tasks on the same CPU from making the per-CPU data inconsistent.

Functions for disabling and enabling migration are presented in Listing 4.13. These functions are implemented by modifying their calling task's affinity. A task calls `migrate_disable()` at the beginning of a critical section and `migrate_enable()` at its end. `migrate_disable()` and `migrate_enable()` increment and decrement `task_struct` member `migration_disabled`, respectively. A value greater than 0 indicates that migration is disabled.

`migrate_disable()` does not immediately modify the caller's affinity mask. Instead, this modification is deferred until the caller is unscheduled in a context switch. As part of the context switch, `migrate_`

```
void migrate_disable(void)
{
    struct task_struct *p = current;

    p->migration_disabled++;
}

void migrate_disable_switch(struct rq *rq, struct task_struct *p)
{
    if (!p->migration_disabled)
        return;
    __do_set_cpus_allowed(p, cpumask_of(rq->cpu), SCA_MIGRATE_DISABLE);
}

void migrate_enable(void)
{
    struct task_struct *p = current;

    if (p->migration_disabled > 1) {
        p-> migration_disabled--;
        return;
    }

    __set_cpus_allowed_ptr(p, &p->cpus_mask, SCA_MIGRATE_ENABLE);
    p->migration_disabled = 0;
}
```

Listing 4.13: Migrate enable and disable.

`disable_switch()` is called, which actually modifies the affinity mask if migration has not since been enabled. It is safe to defer the modification because the caller cannot be migrated while scheduled.

`migrate_disable_switch()` retrieves a pointer to a pre-allocated affinity mask with only the caller's runqueue's current CPU set with `cpumask_of()`. The caller's affinity is changed by calling `__do_set_cpus_allowed()` with flag `SCA_MIGRATE_DISABLE`. This sets the caller's `cpus_ptr` to point to said pre-allocated affinity mask. Note that `__do_set_cpus_allowed()` is called instead of `__set_cpus_allowed_ptr()` because we need not migrate the caller (*i.e.*, call `affine_move_task()`) when disabling migration.

`migrate_enable()` calls `__set_cpus_allowed_ptr()` with a pointer to the caller's non-transient affinity mask `cpus_mask` and flag `SCA_MIGRATE_ENABLE`. This sets `cpus_ptr` back to pointing to `cpus_mask`. Note that it is possible that `cpus_mask` was modified by another call to `__set_cpus_allowed_ptr()` while the `migrate_enable()` caller was mid-critical section. As such, it is

```
void cpuset_attach_task(struct cpuset *cs, struct task_struct *task)
{
    __set_cpus_allowed_ptr(task, cs->effective_cpus, 0);
}

int update_tasks_cpumask(struct cpuset *cs, struct cpumask *new_cpus)
{
    struct task_struct *task;
    for_each_cs_task(task, cs)
        __set_cpus_allowed_ptr()(task, new_cpus, 0);
}
```

Listing 4.14: `cpuset` affinity functions.

necessary to call `__set_cpus_allowed_ptr()` (and thus, `affine_move_task()`) in case affinity was lost for the CPU migration was disabled on.

Unlike `__sched_setaffinity()`, disabling migration does not interact with the ACS.

`cpuset`. The last mechanism for modifying affinity masks is the `cpuset` controller. While an effort has been made previously for listings to resemble the kernel, Listing 4.14 only shares function names with the actual code. The size of the `cpuset` controller code exceeds that of `SCHED_DEADLINE`. Discussing this controller's implementation is outside the scope of this document.

`cpuset_attach_task()` is called when a task is added to a `cpuset` (*i.e.*, writing a TGID to `cgroup.procs`). This calls `__set_cpus_allowed_ptr()` such that `effective_cpus` (a mask that matches `cpuset.cpus.effective`) is written to `task->cpus_mask`. `update_tasks_cpumask()` is called when `cpuset.cpus.effective` is changed (*e.g.*, writing to `cpuset.cpus`, writing to `cpuset.cpus` in a parent `cpuset`, or changing a child `cpuset` into a `root_domain`). `new_cpus` is the new value of `effective_cpus`. `update_tasks_cpumask()` calls `__set_cpus_allowed_ptr()` on each task in the `cpuset`. Note that `for_each_cs_task` is not a real macro in the kernel. Iterating over the tasks in a `cgroup` is fairly complex for reasons that will not be discussed.

Both `cpuset_attach_task()` and `update_tasks_cpumask()` are only called if permitted by the ACS. The ACS checks whether these functions can be called in `cpuset_can_attach()` and `dl_cpuset_cpumask_can_shrink()`, as discussed in Section 4.4.4.

```
void rq_attach_root(struct rq *rq, struct root_domain *rd)
{
    struct sched_class *class;

    if (rq->rd) {
        cpumask_clear_cpu(rq->cpu, rq->rd->span);
        for_each_class(class)
            class->rq_offline(rq);
    }

    rq->rd = rd;
    cpumask_set_cpu(rq->cpu, rd->span);
    for_each_class(class)
        class->rq_online(rq);
}
```

Listing 4.15: Adding CPU to a `root_domain`.

Attaching `root_domains`. Modifying `cpusets` can cause the kernel to rebuild the `root_domains` and `sched_domains`. This process is complex and not described in this document. On a rebuild, CPUs are added to a `root_domain` with `rq_attach_root()` (Listing 4.15).[13]

`rq_attach_root()` clears `rq`'s CPU from the `span` of its old `root_domain`, if it exists (for example, a prior `root_domain` does not exist on boot when CPUs are added to the default `root_domain`). If there was a prior `root_domain`, then for each `sched_class`, `rq_offline()` is called. The CPU is then added to its new `root_domain` `rd`'s `span`, and `rq_online()` is called for each `sched_class`.

The prototypes of `rq_online()` and `rq_offline()` are as follows.

```
void (*rq_online)(struct rq *rq);
void (*rq_offline)(struct rq *rq);
```

These functions alert the scheduler of their corresponding class that tasks should (`rq_online()`) or should not (`rq_offline()`) be scheduled on the CPU of `rq`. Besides being called in `rq_attach_root()`, these functions are also called when CPUs are activated or deactivated. `SCHED_DEADLINE` uses these functions to manage `SCHED_DEADLINE`-specific data stored in the `root_domains`.

[13] CPU hotplug logic is being omitted. We are assuming that any runqueues `rq_attach_root()` is called on correspond with CPUs that are *online*. With hotplug, the set of online CPUs is dynamic. Depending on whether a given CPU is online or offline, `rq_offline()` and `rq_online()` may not need to be called.

4.3.8 Stop Class

stop_sched_class maintains a task and a FIFO queue on each CPU. Each queue is a linked-list containing callback functions. Whenever one of these queues becomes non-empty, the corresponding stop_sched_class task wakes up and executes the callback functions in the queue. Because these tasks belong to stop_sched_class, the highest priority sched_class, these callback functions execute with higher priority than any other task. These tasks then block once their corresponding queues are empty.

Tasks of stop_sched_class are used by the scheduler for migrating tasks of the user-level sched_classes. The main use case is migrating a task that is already scheduled on a CPU. To keep scheduling data structures consistent, such a task needs to be unscheduled from its CPU before it can be migrated. The scheduler unschedules the target task by enqueuing a callback function onto the CPU's FIFO queue, thereby waking the CPU's stop_sched_class task that preempts the target task. The callback function enqueued by the scheduler then performs the migration.

A callback function is enqueued with stop_one_cpu().

```
int stop_one_cpu(unsigned int cpu, cpu_stop_fn_t fn, void *arg)
```

cpu is the target CPU, fn is a callback function pointer of type cpu_stop_fn_t (*i.e.*, fn must return an int and take a void pointer), and arg is the argument to be passed to fn when it is called. stop_one_cpu() returns the return value of fn. As such, stop_one_cpu() blocks until fn completes. More commonly used by the scheduler is the non-blocking variant stop_one_cpu_nowait().

```
bool stop_one_cpu_nowait(unsigned int cpu, cpu_stop_fn_
    t fn, void *arg, struct cpu_stop_work *work_buf)
```

The non-blocking variant requires an additional argument work_buf that is a pointer to a cpu_stop_work (nodes in the aforementioned linked-list queues used by stop_sched_class are of type cpu_stop_work). work_buf is then a pointer to an allocated region of memory that will store the node that will correspond to calling fn on cpu. Because most members of cpu_stop_work are assigned values within stop_one_cpu_nowait() (based on the other arguments to stop_one_cpu_nowait()), we do not describe these members here.

One migration function used as an argument of stop_one_cpu_nowait() is push_cpu_stop() (Listing 4.17), which pushes a target task from the stopped CPU's runqueue. We highlight push_cpu_stop() because it calls class-specific function find_lock_rq() and because it is used by SCHED_

DEADLINE. The following is a high-level example of how SCHED_DEADLINE may use push_cpu_stop().

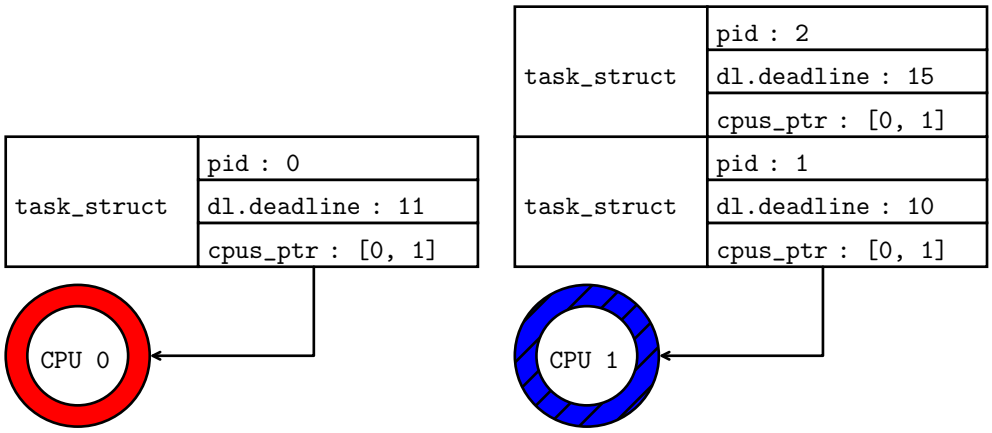
▼ **Example 4.3.** Consider a system with two CPUs as illustrated in Figure 4.5a. Initially, there are three SCHED_DEADLINE tasks queued on these CPUs' runqueues. Task 0 is scheduled on CPU 0 and Task 1 is scheduled on CPU 1.

Task 0 calls migrate_disable(), fixing itself on CPU 0. migrate_disable_switch() sets cpus_ptr to point to a mask with only CPU 0 set (Figure 4.5b) when Task 0 is next preempted (Figure 4.5c). When Task 3 wakes, Task 3 is enqueued on CPU 0 due to Task 0 having a later deadline than Task 1 on CPU 1.

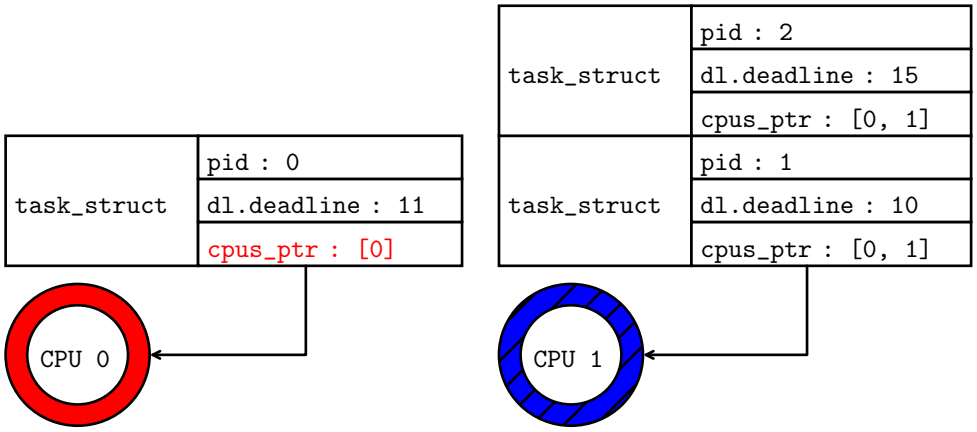
Suppose Task 1 on CPU 1 suspends (Figure 4.5d). This suspension triggers CPU 1 to reschedule, during which Task 1 is dequeued. SCHED_DEADLINE wishes to execute Tasks 0 and 3, the highest priority tasks in the system, on the two CPUs. Tasks 0 and 3 are both on CPU 0's runqueue. Ordinarily, Task 0 would be migrated to CPU 1's runqueue prior to its rescheduling (*i.e.*, during balance()), but Task 0 has disabled migration. Task 3, on the other hand, could be migrated to CPU 1, but is currently scheduled on CPU 0.

balance() observes that the task it wants to pull (Task 0) has disabled migration. It calls stop_one_cpu_nowait() to wake the stopper task on CPU 0 (Figure 4.5e). This preempts Task 3, making it migratable. push_cpu_stop(), queued on the stopper task by stop_one_cpu_nowait(), migrates Task 3 to CPU 1 (Figure 4.5f). After push_cpu_stop() completes and the stopper task on CPU 0 suspends (Figure 4.5g), Task 0 is the highest-priority task on CPU 0 and Task 3 is the highest-priority task on CPU 1. Thus, SCHED_DEADLINE is able to schedule the highest-priority tasks in the system. ▲

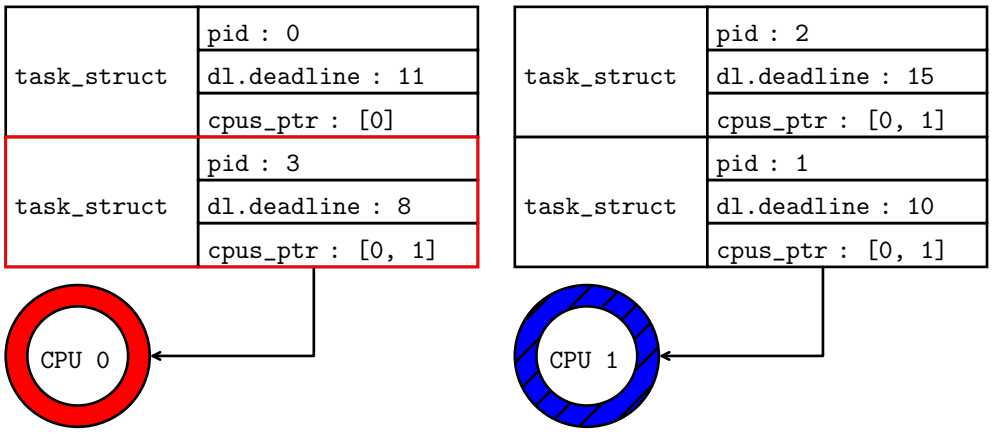
Queueing push_cpu_stop() for runqueue rq is usually prepared for by calling get_push_task() (Listing 4.16). get_push_task() returns the task to be pushed by push_cpu_stop(). Recall from Example 4.3 how this is the currently scheduled task on the target runqueue rq. Before returning the current task on rq, p, get_push_task() first confirms that rq->push_work is not already being used in another push by checking push_busy. push_work is the per-runqueue linked-list node of type cpu_stop_work that stores push_cpu_stop() on rq's stopper queue. push_busy is a boolean that indicates whether push_work is in use. get_push_task() also checks that p is actually migratable



(a) Initial system.

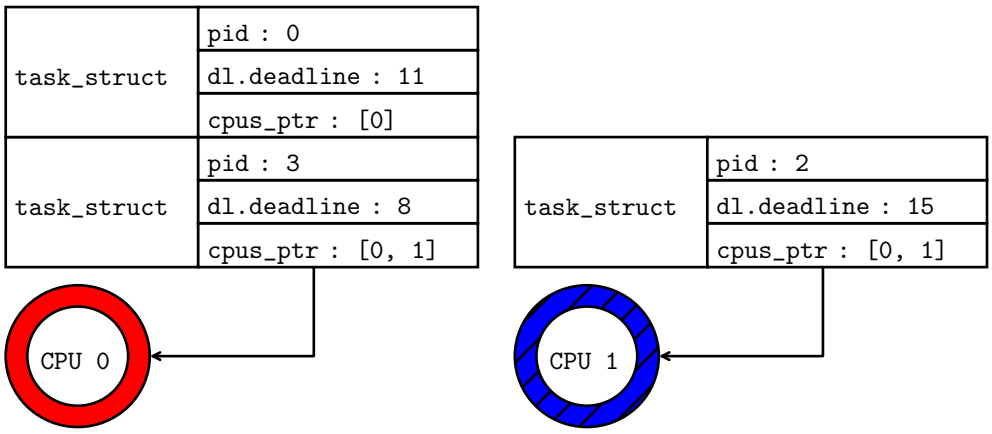


(b) Task 0 calls `migrate_disable()`.

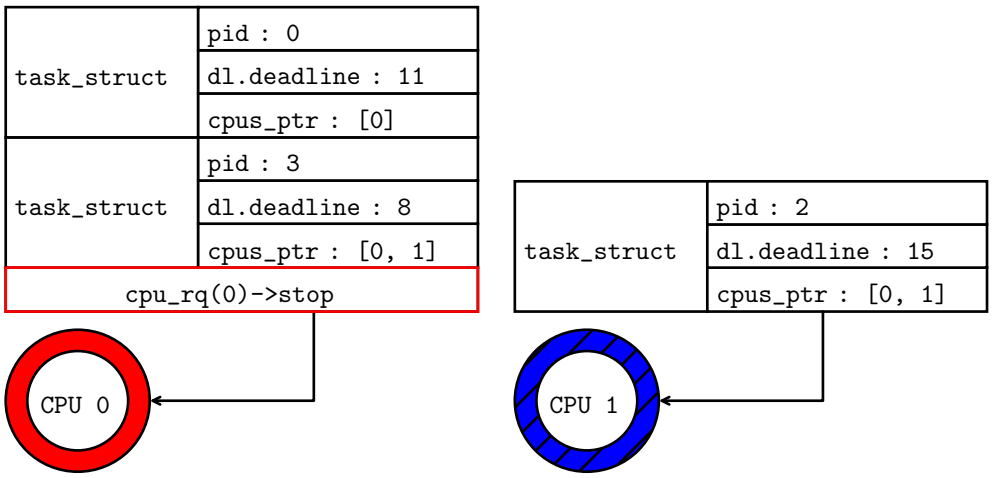


(c) Task 3 wakes and preempts Task 0.

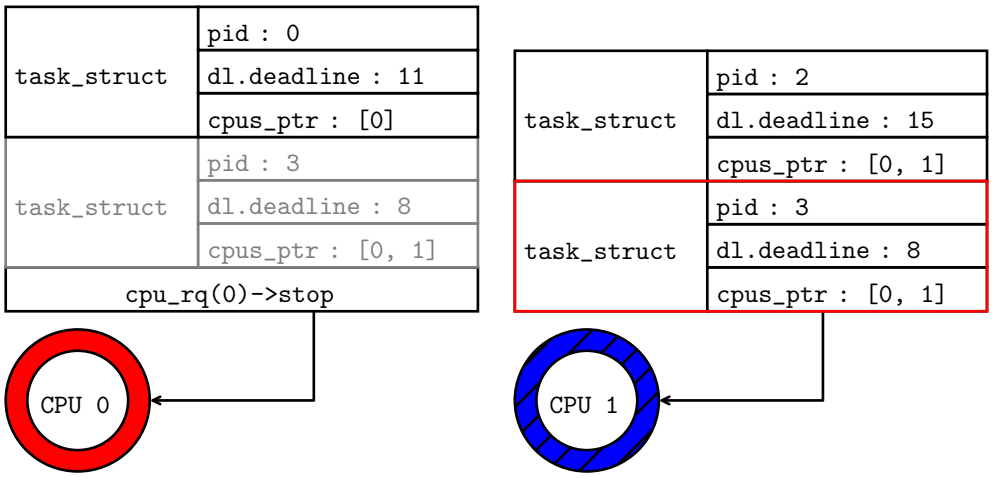
Figure 4.5: `push_cpu_stop()` example.



(d) Task 1 suspends and CPU 1 must reschedule.

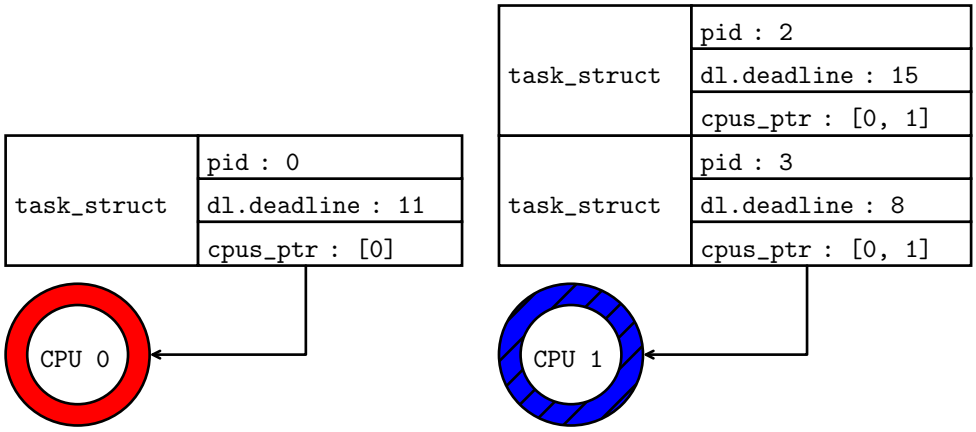


(e) CPU 1 calls `stop_one_cpu_nowait()` on CPU 0.



(f) `push_cpu_stop()` pushes Task 0 to CPU 1.

Figure 4.5: `push_cpu_stop()` example (continued).



(g) `cpu_rq(0)->stop` suspends.

Figure 4.5: `push_cpu_stop()` example (continued).

```
struct task_struct *get_push_task(struct rq rq)
{
    struct task_struct *p = rq->curr;
    if (rq->push_busy)
        return NULL;
    if (p->nr_cpus_allowed == 1)
        return NULL;
    if (p->migration_disabled)
        return NULL;
    rq->push_busy = true;
    return p;
}
```

Listing 4.16: `get_push_task()`.

(*i.e.*, p->nr_cpus_allowed exceeds one and p has not disabled migration). If push_work is available and p is migratable, then push_busy is set and p is returned. push_cpu_stop() is then queued as follows.

```
stop_one_cpu_nowait(rq->cpu, push_cpu_stop, p, &rq->push_work);
```

Code for push_cpu_stop() is presented in Listing 4.17. rq is set to point to the runqueue of the stop task by the this_rq() macro that returns a pointer to the runqueue of the executing CPU (recall that the stop task on a CPU never migrates). Argument arg, which is a void pointer to match cpu_stop_fn_t, is casted into task_struct pointer p, the target task to push.

During the delay between push_cpu_stop() getting enqueued on a stopper task's queue and said stopper task executing push_cpu_stop(), the state of the scheduler may have changed. As such, push_cpu_stop() may bail on migrating p (*i.e.*, with goto out_unlock). At line 8, push_cpu_stop() checks that rq is still p's runqueue, and quits if it is not.

At line 11, push_cpu_stop() checks that p did not disable migration, and quits if it has. If migration was disabled, push_cpu_stop() sets MDF_PUSH in migration_flags. Flag MDF_PUSH indicates that p should be pushed once migrate_enable() is called. migrate_enable() calls affine_move_task(), which is responsible for checking migration_flags and performing this push if MDF_PUSH is set. On the other hand, if migration is still enabled, MDF_PUSH is unset in migration_flags.

At line 17, push_cpu_stop() calls class-specific function find_lock_rq(). The prototype of find_lock_rq() is as follows.

```
struct rq *(*find_lock_rq)(struct task_struct *p, struct rq *rq);
```

find_lock_rq() finds a runqueue that p should be pushed to or NULL if no such runqueue can be found. rq is a runqueue that p is currently queued on. The functionality of find_lock_rq() is similar to that of select_task_rq(), with both functions calling the same helper functions in rt_sched_class and dl_sched_class (find_lock_rq() is not implemented in fair_sched_class). Besides returning an rq pointer instead of an int, find_lock_rq() differs from select_task_rq() by locking the spinlock of the returned runqueue. When find_lock_rq() returns, both the locks of the rq and the returned runqueue are held.

If find_lock_rq() fails to return a runqueue, push_cpu_stop() again quits on migrating p.

```
1  int push_cpu_stop(void *arg)
2  {
3      struct rq *lowest_rq = NULL, *rq =  this_rq();
4      struct task_struct *p = arg;
5      struct sched_class *class = p->sched_class;
6      int dest_cpu;
7
8      if (task_rq(p) != rq)
9          goto out_unlock;
10
11     if (p->migration_disabled) {
12         p->migration_flags |= MDF_PUSH;
13         goto out_unlock;
14     }
15     p->migration_flags &= ~MDF_PUSH;
16
17     lowest_rq = class->find_lock_rq(p, rq);
18     if (!lowest_rq)
19         goto out_unlock;
20     dest_cpu = lowest_rq->cpu;
21
22     if (task_rq(p) == rq) {
23         p->on_rq = TASK_ON_RQ_MIGRATING;
24         class->dequeue_task(rq, p, 0);
25         class->migrate_task_rq(p, dest_cpu);
26         __set_task_cpu(p, dest_cpu);
27         class->enqueue_task(lowest_rq, p, ENQUEUE_MIGRATED);
28         p->on_rq = TASK_ON_RQ_QUEUED;
29         resched_curr(lowest_rq);
30     }
31
32 out_unlock:
33     rq->push_busy = false;
34     return 0;
35 }
```

Listing 4.17: Pushing with `stop_sched_class`.

At line 22, `push_cpu_stop()` again checks that `rq` is the runqueue of `p`. This may have changed since line 8 due to `find_lock_rq()` potentially dropping `rq`'s lock in order to acquire `lowest_rq`'s lock in CPU index order. If `rq` is still the runqueue of `p`, the task is migrated using `dequeue_task()`, `migrate_task_rq()`, *etc.* (recall the discussion of these functions in Sections 4.3.1-4.3.2).

After `push_cpu_stop()` either finishes migrating or decides to bail out, at line 32, `rq->push_busy` is unset. This indicates that `rq->push_work` can now be reused. `push_cpu_stop()` always returns 0.

Stopping multiple CPUs. There also exist functions for simultaneously waking the `stop` tasks of multiple CPUs. These include functions `stop_two_cpus()` and `stop_machine()`, which wake the `stop` task on two and all CPUs, respectively. Though neither of these functions is used by `SCHED_DEADLINE`, we highlight it here because it is used in the patch that will be discussed in Section 5.3.

```
int stop_two_cpus(unsigned int cpu1, unsigned int cpu2, cpu_stop_fn_
    t fn, void *arg)
```

Though the `stop` task is woken on both CPUs, note that `fn` is only executed by the `stop` task on `cpu1`. The `stop` task on `cpu2` spins until `fn` returns.

4.4 SCHED_DEADLINE

Having covered the common infrastructure, this section covers how `SCHED_DEADLINE` uses its `sched_class` and data structures to implement EDF. Going forward, assume all tasks are `SCHED_DEADLINE` tasks unless stated otherwise.

4.4.1 Data Structures

We start by discussing `SCHED_DEADLINE`-specific data structures.

`sched_dl_entity`. Recall that the `dl_sched_class` scheduling entity structure is the `sched_dl_entity` (Listing 4.18). A `sched_dl_entity` represents the CBS that encapsulates the `task_struct` containing the `sched_dl_entity`.

`rb_node` is the red-black tree node used to add the corresponding `sched_dl_entity` to a `SCHED_DEADLINE` sub-runqueue `dl_rq`.

```
struct sched_dl_entity {
    struct rb_node          rb_node;

    u64                     dl_runtime;
    u64                     dl_deadline;
    u64                     dl_period;
    u64                     dl_bw;
    u64                     dl_density;

    s64                     runtime;
    u64                     deadline;
    unsigned int            flags;

    unsigned int            dl_throttled;
    unsigned int            dl_yielded;
    unsigned int            dl_non_contending;

    struct hrtimer          dl_timer;

    struct hrtimer          inactive_timer;

    struct sched_dl_entity  *pi_se;
};
```

Listing 4.18: `struct sched_dl_entity`.

`dl_runtime`, `dl_deadline`, and `dl_period` represent the maximum budget, deadline, and period of the CBS. `dl_bw` and `dl_density` are derived from these parameters. Both are fractions that are inflated by left-shifting their numerators by `BW_SHIFT` (*i.e.*, 20) before division. This allows the kernel to represent bandwidths and densities as integers, thereby avoiding floating point arithmetic. `dl_bw` is (`dl_runtime` $\ll$ `BW_SHIFT`)/`dl_period` and `dl_density` is (`dl_runtime` $\ll$ `BW_SHIFT`)/`dl_deadline`. Note that all bandwidth quantities in `SCHED_DEADLINE` are similarly left-shifted by `BW_SHIFT`.

`runtime` and `deadline` represent the current budget and absolute deadline of the CBS.

`flags` stores the `SCHED_DEADLINE` task flags set by `__sched_setscheduler()` (*e.g.*, `SCHED_FLAG_RECLAIM`, `SCHED_FLAG_DL_OVERRUN`, and `SCHED_FLAG_SUGOV`).

`dl_throttled` and `dl_yielded` are flags indicating whether the CBS is currently unschedulable due to being throttled or yielded. `dl_non_contending` is a flag used by GRUB (discussed in Section 4.4.8).

`dl_timer` and `inactive_timer` are `hrtimer`s. `dl_timer` is armed when a CBS is throttled, and fires upon the next period to replenish the CBS's budget. `inactive_timer` is armed when a task suspends while still active (*i.e.*, is before its zero-lag time), and fires when the task becomes inactive (*i.e.*, at the

```
struct dl_rq {
    struct rb_root_cached   root;

    unsigned int            dl_nr_running;

    struct {
        u64                 curr;
        u64                 next;
    } earliest_dl;

    unsigned int            dl_nr_migratory;
    int                     overloaded;

    struct rb_root_cached   pushable_dl_tasks_root;

    u64                     running_bw;

    u64                     this_bw;
    u64                     extra_bw;
    u64                     max_bw;
    u64                     bw_ratio;
};
```

Listing 4.19: `struct dl_rq`.

zero-lag time). For each sub-runqueue `dl_rq`, the `inactive_timer` helps track `dl_rq->running_bw`, the total bandwidth of active tasks corresponding with `dl_rq`, which is used for GRUB (discussed in Section 4.4.8) and DVFS (discussed in Section 4.4.9).

`pi_se` points to the `sched_dl_entity` the corresponding task `p` is inheriting `p->prio` from. When not inheriting, `pi_se` points to its containing `sched_dl_entity`.

`dl_rq`. The sub-runqueue of `dl_sched_class` is the `dl_rq` (Listing 4.19).

`root` is the root of the deadline-ordered red-black tree of queued `sched_dl_entity`s on the containing `struct rq`. A `sched_dl_entity` is added to `root` with its `rb_node` member.

`dl_nr_running` is the number of `sched_dl_entity`s on `root`. Note that throttled `sched_dl_entity`s are not in this tree and do not count towards `dl_nr_running`. The `sched_dl_entity`s of non-`SCHED_DEADLINE` tasks that have inherited `SCHED_DEADLINE` priorities are in this tree and count towards `dl_nr_running`.

`earliest_dl.curr` is the earliest deadline of any `sched_dl_entity` on `root`. `earliest_dl.next` is the earliest deadline of any `sched_dl_entity` on `pushable_dl_tasks_root` (to be discussed shortly). `earliest_dl.curr` is 0 if `root` is empty (*i.e.*, `dl_nr_running == 0`). The

deadlines in earliest_dl are used by SCHED_DEADLINE when deciding whether or not to pull a task from another dl_rq (*i.e.*, earliest_dl.next of the other dl_rq should be earlier than earliest_dl.curr of the pulling dl_rq). Note that earliest_dl.next may not be 0 if pushable_dl_tasks_root is empty.

dl_nr_migratory is the number of sched_dl_entitys on root such that the task_structs that contain these sched_dl_entitys have nr_cpus_allowed > 1.

overloaded is a boolean that is equivalent to dl_nr_migratory!= 0 && dl_nr_running > 0. overloaded indicates whether or not other runqueues should attempt to pull tasks from this dl_rq. overloaded matches the bit corresponding with the dl_rq's CPU in the root_domain's dlo_mask.

pushable_dl_tasks_root is another deadline-ordered red-black tree root. A task_struct is in pushable_dl_tasks_root if its contained sched_dl_entity is in root, nr_cpus_allowed > 1, and it is not currently scheduled. A task_struct is added to pushable_dl_tasks_root with its pushable_dl_tasks member. Note that because the scheduled task is never on pushable_dl_tasks_root, dl_nr_migratory may not be the number of tasks in this tree.

running_bw is the total bandwidth of active tasks p such that task_rq(p) == rq, where rq is the containing struct rq of this dl_rq. running_bw is used by either GRUB (Section 4.4.8) or DVFS (Section 4.4.9), both of which, as running_bw decreases, scale down the rate of budget consumption for tasks executing on the corresponding CPU. Note that throttled tasks and active suspended tasks contribute to running_bw despite not being on the dl_rq.

this_bw, extra_bw, max_bw, and bw_ratio are additional parameters used by GRUB. These parameters will be discussed in Section 4.4.8.

dl_bw. The dl_bw stores the total bandwidth of tasks on the CPUs in its containing root_domain. Its members are as follows.

```
struct dl_bw {
    u64 bw;
    u64 total_bw;
};
```

bw is the fraction sched_rt_runtime_us/sched_rt_period_us or −1 if −1 is written to sched_rt_runtime_us (*i.e.*, the ACS is disabled). bw represents the fraction of the containing root_domain's

CPU capacity that SCHED_DEADLINE tasks are permitted to consume by the ACS. Because sched_rt_runtime_us and sched_rt_period_us are set globally, bw has the same value in every dl_bw. total_bw is the sum of the dl_bws of the sched_dl_entitys executing on the CPUs in the root_domain. Re-stating (4.1), the primary purpose of the ACS is to maintain for each root_domain that

$$\texttt{dl\_bw.total\_bw} \leq \texttt{dl\_bw.bw} \cdot \text{total capacity}, \tag{4.2}$$

where total capacity is the sum of the capacities of the root_domain's CPUs (how capacities are derived will be discussed when asymmetric capacities are discussed in Section 4.4.6). Note that because both bw and total_bw are bandwidth quantities, bw and total_bw are actually left-shifted by BW_SHIFT. This does not affect (4.2) because both sides are scaled by the same factor.

cpudl. A cpudl (Listing 4.20) stores information about the CPUs in the containing root_domain's span. A cpudl makes migration decisions between CPUs in its root_domain more efficient. An example of a cpudl will be presented after its members are described. The primary member of cpudl is elements, an array-based heap of a subset of the CPUs in span. Only CPUs with queued sched_dl_entitys on their dl_rqs are in elements. The remaining CPUs in span are in mask free_cpus.

The key of each CPU in elements is the earliest deadline of any task on said CPU's dl_rq. elements is a max-heap, thus the CPU with the latest of these deadline keys is at the root. cpudlsize is the number of CPUs in elements, and is used for inserting CPUs into elements. Each element of the array elements is of type struct cpudl_item. dl and cpu are the deadline key and CPU index of the CPU represented at that index in elements.

idx is less straightforward. elements is actually two equally-sized arrays interleaved with each other. One array is the aforementioned heap of CPUs (*i.e.*, dl and cpu in cpudl_item). The other array is a mapping from a CPU index to the index (idx in cpudl_item) in the heap array such that cpu matches said CPU index.

▼ **Example 4.4.** Consider a root_domain with six CPUs indexed 0-5. The earliest-deadline task on CPU 0's dl_rq has deadline 5, CPU 1's has deadline 1, CPU 2's has deadline 3, CPU 3's has deadline 9, CPU 4's has deadline 8, and 5's has deadline 7. The max-heap of these CPUs is illustrated in Figure 4.6a.

```
struct cpudl_item {
    u64 dl;
    int cpu;
    int idx;
};

struct cpudl {
    int                 cpudlsize;
    cpumask_var_t       free_cpus;
    struct cpudl_item   *elements;
};
```

Listing 4.20: `struct cpudl`.

Observe how CPU 3 is at the root of the heap due to having the latest of the earliest (on each CPU's `dl_rq`) deadlines.

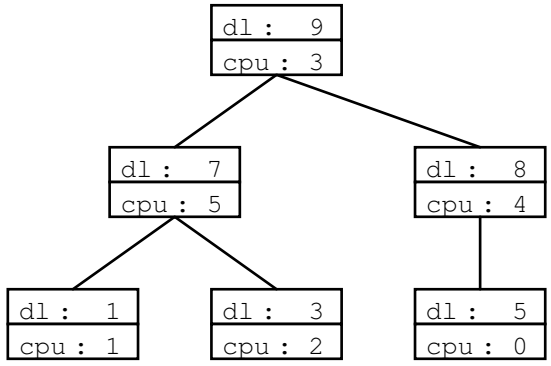
The heap in Figure 4.6a can be stored as an array as in the left side of Figure 4.6b. The data at index 0 in this array corresponds with the root in the heap. The children of a node at index `i` are found at indices `i << 1 + 1` and `i << 1 + 2` (*e.g.*, the children of the node at index 1, which has `cpu: 5` and `dl: 7`, are at indices `1 << 1 + 1 == 3` and `1 << 1 + 2 == 4`).

The array on the right of Figure 4.6b is a mapping. The mapping array maps a given CPU index to the index of the node in the heap array that corresponds to said CPU. For example, the 0^{th} item in the mapping array has `idx: 5`. This means the node with `cpu: 5` is found at `idx: 5` in the heap array.

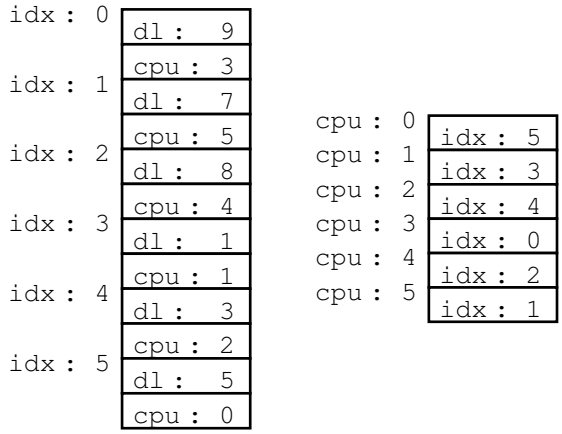
Figure 4.6c shows the interleaving of the two arrays. The first node in the heap array (`dl: 9` and `cpu: 3`) is followed by the first index in the mapping array (`idx: 5`), the second node in the heap (`dl: 7` and `cpu: 5`) is followed by the second index in the mapping (`idx: 3`), *etc.* This interleaved array is how the heap and mapping are stored in `elements`. ▲

4.4.2 Multiprocessor Scheduling

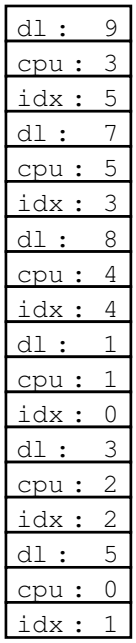
This subsection discusses how tasks are migrated between runqueues to schedule those with the earliest deadlines. For now, we assume that each task's deadline is some arbitrary constant. Actual deadlines are determined by the CBS implementation and will be discussed in Section 4.4.3. Migration logic can be discussed independently of the CBS because migrations caused by the suspending and waking of tasks are analogous to migrations caused by the throttling and replenishment of tasks.



(a) `cpudl` heap illustration.



(b) Array representation of heap (left) and `cpu` to `idx` mapping (right).



(c) `elements`.

Figure 4.6: `cpudl` illustration.

The basic building blocks of migrations are enqueuing a task onto the dl_rq being migrated to and dequeuing a task from its current dl_rq. The enqueues and dequeues that result in the earliest-deadline tasks being on different runqueues primarily occur due to pushes and pulls. Ignoring CBS throttling and replenishment, these pushes and pulls are mostly triggered by tasks suspending and waking, possibly on a different dl_rq.

4.4.2.1 Enqueuing and Dequeuing

Consider Listing 4.21. Deferring discussion of CBS for Section 4.4.3 and GRUB for Section 4.4.8, the function of enqueue_task_dl() and dequeue_task_dl() is to insert and remove task p from dl_rq rq->dl's red-black trees. These trees are the one rooted at rq->dl.root (all runnable tasks on the dl_rq) and at rq->pushable_dl_tasks_root (the subset of migratable tasks). __enqueue_dl_entity() and __dequeue_dl_entity() insert and remove p onto root (with rb_node p->dl.rb_node). enqueue_pushable_dl_task() and dequeue_pushable_dl_task() are the analogous insertion and deletion functions for pushable_dl_tasks_root (with rb_node p->pushable_dl_tasks). These four helper functions are also where a majority of the state of dl_rqs and the SCHED_DEADLINE-relevant state of root_domains are maintained. __enqueue_dl_entity() and __dequeue_dl_entity() update, in the dl_rq rq->dl, members dl_nr_running, earliest_dl.curr, dl_nr_migratory, and overloaded, and, in the root_domain rq->rd, members cpudl and dlo_mask. enqueue_pushable_dl_task() and dequeue_pushable_dl_task() update earliest_dl.next.

There is a nuance to be discussed about Listing 4.21. It may be unclear why dequeue_task_dl(), before calling dequeue_pushable_dl_task(), does not need to perform the same check that enqueue_task_dl() does on line 8 to determine if p is migratable before calling enqueue_pushable_dl_task(). This is because the scheduler maintains that the parent pointer of any unqueued rb_node points back to itself. This allows dequeue_pushable_dl_task() (and also __dequeue_dl_entity()) to first check the relevant rb_node (*e.g.*, p->pushable_dl_tasks or p->dl.rb_node), and return immediately if p is not actually queued.

Because these dequeue helper functions first check that a task is actually queued, it is generally safe to call dequeue_task_dl() even if p is not actually on the dl_rq. This is necessary because there are instances where a task p may be on a runqueue rq (from the perspective of the common scheduler

```
1  void enqueue_task_dl(struct rq *rq, struct task_struct *p, int flags)
2  {
3      /* Do GRUB logic */
4
5      /* Do CBS logic */
6
7      __enqueue_dl_entity(&p->dl);
8      if (rq->curr != p && p->nr_cpus_allowed > 1)
9          enqueue_pushable_dl_task(rq, p);
10 }
11
12 void dequeue_task_dl(struct rq *rq, struct task_struct *p, int flags)
13 {
14     __dequeue_dl_entity(&p->dl);
15     dequeue_pushable_dl_task(rq, p);
16
17     /* Do GRUB logic */
18 }
```

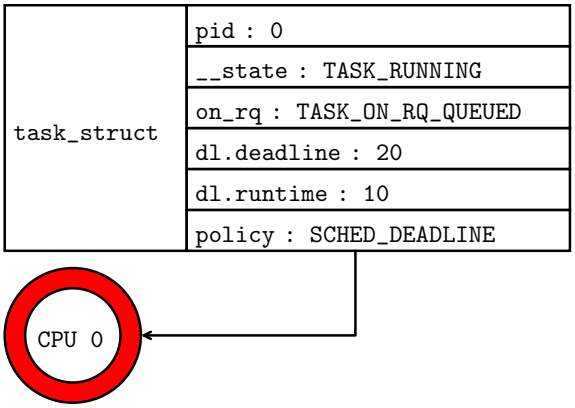
Listing 4.21: `enqueue_task_dl()` and `dequeue_task_dl()`.

infrastructure) while *not* being on the corresponding `dl_rq`. This can occur when `p` is not suspended (*i.e.*, `p->on_rq` is `TASK_ON_RQ_QUEUED`) but is throttled (hence, not on the `dl_rq`). Because CBS throttling has not yet been discussed, we illustrate the necessity of being able to call `dequeue_task_dl()` on such tasks with a high-level example.

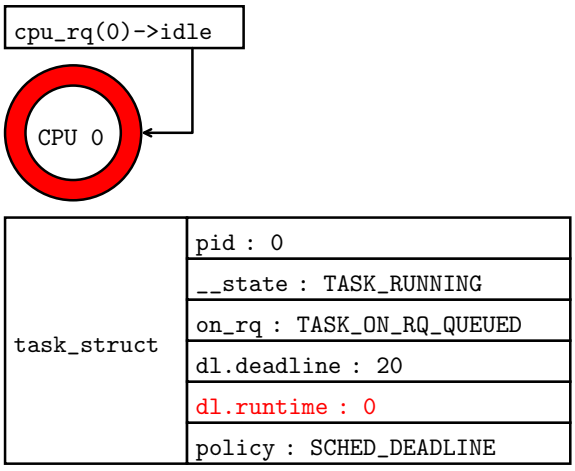
▼ **Example 4.5.** Consider Task 0 illustrated in Figure 4.7a on CPU 0's runqueue. Suppose at some point Task 0 exhausts its budget, `dl.runtime`. Due to being unrunnable due to lack of budget, `SCHED_DEADLINE` dequeues Task 0 from CPU 0's runqueue (Figure 4.7b). Despite being dequeued, Task 0 is not suspended due to sleeping or being blocked on some resource. Thus, from the perspective of the common scheduling infrastructure, Task 0 still appears runnable (*i.e.*, `__state` is `TASK_RUNNING` and `on_rq` is `TASK_ON_RQ_QUEUED`).

Now consider if a request is made to change the policy of Task 0 to `SCHED_FIFO` while it is throttled. Then the change pattern (recall Listing 4.7) observes that `on_rq` is `TASK_ON_RQ_QUEUED` and calls `dequeue_task_dl()` on Task 0. Because `__dequeue_dl_entity()` and `dequeue_pushable_dl_task()` can observe that Task 0 is not enqueued, the call to `dequeue_task_dl()` returns without incorrectly attempting to dequeue Task 0 again.

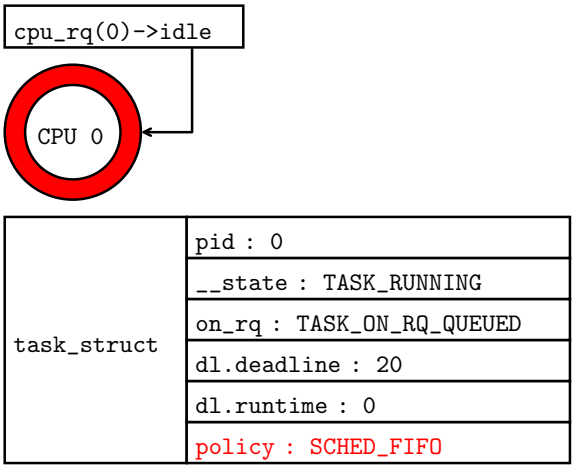
After Task 0's policy is changed (Figure 4.7c), because `on_rq` was observed to be `TASK_ON_RQ_QUEUED`, the change pattern calls `rt_sched_class`'s `enqueue_task()` function. This places



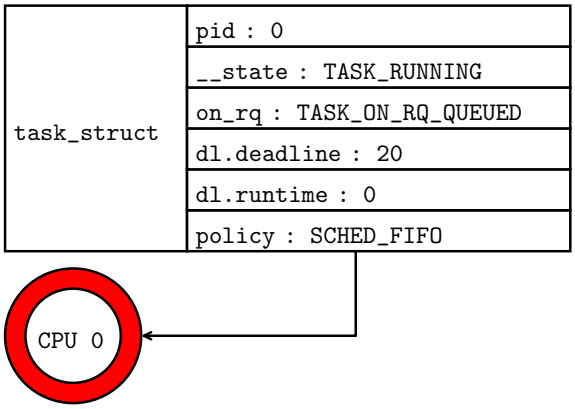
(a) Initial system.



(b) Task 0 throttled for exhausting budget.



(c) Task 0 switches to `SCHED_FIFO`.



(d) Task enqueued in change pattern.

Figure 4.7: Class change of throttled task.

Task 0 back on CPU 0's runqueue. This enqueue is necessary for correct scheduling; now that Task 0 has left SCHED_DEADLINE, the scheduler does not care that runtime is exhausted. ▲

Note that similarly to how dequeue_task_dl(), when called from the change pattern, must not dequeue an already throttled task, enqueue_task_dl() must not accidentally enqueue a throttled task when called from the change pattern. enqueue_task_dl() checks dl.dl_throttled to determine if the task should actually be enqueued. This falls under CBS logic that will be discussed in Section 4.4.3.

4.4.2.2 Pushes and Pulls

A migration dequeues a task from one runqueue and enqueues it on another runqueue. Most migrations in SCHED_DEADLINE occur due to pushes and pulls. We start with pulls because they are less complex than pushes.

Pulls. A pull migrates a high-priority task from a source runqueue onto the pulling runqueue. pull_dl_task() is presented in Listing 4.22.

A majority of pull_dl_task() is the for loop beginning at line 9. for_each_cpu() iterates over the CPUs in dlo_mask using cpu as the iteration variable. Recall from the discussion of root_domains in Section 4.2 that dlo_mask contains the CPUs in the corresponding root_domain with spare (*i.e.*, at least one unscheduled and migratable) tasks. pull_dl_task() iterates over these CPUs to pull such spare tasks.

For the CPU cpu of a given iteration of the for loop, pull_dl_task() first checks if an attempt should be made to pull from cpu. The remainder of the iteration is skipped if not. This check compares earliest_dl.curr of this_rq->dl, the earliest deadline of any task on the pulling dl_rq, against earliest_dl.next of src_rq->dl, the earliest deadline of a migratable task on cpu's dl_rq. If the deadline corresponding with the pulling dl_rq precedes that of cpu's dl_rq, a pull is not attempted. Note that earliest_dl.next is oblivious as to whether the task with its deadline actually has affinity for this_cpu.

Function pick_earliest_pushable_dl_task() iterates over the tasks in the tree rooted at src_rq->dl.pushable_dl_tasks_root to find the earliest-deadline task p such that p has affinity for this_cpu (specifically, this_cpu is set in p->cpus_mask). If no such task is on the tree, pick_earliest_pushable_dl_task() returns NULL.

```
1  void pull_dl_task(struct rq *this_rq)
2  {
3      int this_cpu = this_rq->cpu, cpu;
4      struct task_struct *p, *push_task = NULL;
5      bool resched = false;
6      struct rq src_rq;
7      u64 dmin = LONG_MAX;
8
9      for_each_cpu(cpu, this_rq->rd->dlo_mask) {
10         src_rq = cpu_rq(cpu);
11
12         /* Check should pull from src_rq */
13
14         p = pick_earliest_pushable_dl_task(src_rq, this_cpu);
15
16         if (p && p->dl.deadline < dmin &&
17             (!this_rq->dl.dl_nr_running ||
18                 p->dl.deadline < this_rq->dl.earliest_dl.curr))
19         {
20             if (p->migration_disabled)
21                 push_task = get_push_task(src_rq);
22             else {
23                 p->on_rq = TASK_ON_RQ_MIGRATING;
24                 dequeue_task_dl(src_rq, p, 0);
25                 migrate_task_rq_dl(p, this_cpu);
26                 __set_task_cpu(p, this_cpu);
27                 enqueue_task_dl(this_rq, p, ENQUEUE_MIGRATED);
28                 p->on_rq = TASK_ON_RQ_QUEUED;
29
30                 dmin = p->dl.deadline;
31                 resched = true;
32             }
33         }
34
35         if (push_task) {
36             stop_one_cpu_nowait(src_rq->cpu, push_cpu_
                    stop, push_task, &src_rq->push_work);
37             push_task = NULL;
38         }
39     }
40
41     if (resched)
42         resched_curr(this_rq);
43 }
```

Listing 4.22: Pull task pseudocode.

The condition starting at line 16 determines if a task is migrated from `src_rq` to `this_rq`. This condition is not redundant with respect to the aforementioned checks (line 12) for two reasons. First, the earlier checks referred to `earliest_dl.next`, which is oblivious to affinities, while the condition at line 16 refers to `p`, which is guaranteed to have affinity for `this_cpu` by `pick_earliest_pushable_dl_task()`. Second, the earlier checks are done without holding `src_rq`'s spinlock, while this lock is acquired by line 16.

At line 16, `pull_dl_task()` checks that `pick_earliest_pushable_dl_task()` returned a task (`p`), that `p` has an earlier deadline than any task pulled in an earlier iteration (`p->dl.deadline < dmin`), and either `this_rq` has no tasks (`!this_rq->dl.dl_nr_running`) or `p` has an earlier deadline than any task on `this_rq` (`p->dl.deadline < this_rq->dl.earliest_dl.curr`).

If the condition at line 16 is `true`, then `pull_dl_task()` checks if `p`, the to-be-pulled task, has disabled migration. If migration is disabled, then the stop class is used to migrate `src_rq->curr` (recall Example 4.3). `push_cpu_stop()` is queued on `src_rq` on line 36. Otherwise, if migration is enabled, `p` is migrated to `this_rq` (lines 23-28). `resched` is set to `true` to indicate that `this_cpu` should reschedule (line 41) due to an earlier-deadline task migrating to `this_rq`.

Pushes. `push_dl_task()`, which sends a task from a pushing runqueue to a runqueue with either no `SCHED_DEADLINE` tasks or only tasks with deadlines later than that of the pushed task's deadline, is more complicated than `pull_dl_task()`. The runqueue being pushed to is called a *later* runqueue. Instead of iterating over CPUs, as done by `pull_dl_task()` to select which runqueue to pull from, `push_dl_task()` checks the `cpudl` to determine the later runqueue to push to. This checking of the `cpudl` is done by helper function `find_lock_later_rq()`, which itself calls function `find_later_rq()`. `find_later_rq()` takes a `task_struct` pointer `p` as its sole argument and returns the CPU index of the later runqueue (or `-1` if no later runqueue can be found). When `find_later_rq()` returns a CPU index that is not `-1`, the CPU is always set in `p->cpus_mask`.

The complexity of `find_later_rq()` makes presenting pseudocode impractical. The return value of `find_later_rq()` is best understood when broken down into cases. For pushing runqueue `rq` and pushed task `p`, the return value of `find_later_rq()` is as follows.

1. The intersection of `p->cpus_mask` and `rq->cpudl.free_cpus` has CPUs.

    (a) The system contains CPUs with asymmetric capacities.

Discussion of this case is deferred to Section 4.4.6.

(b) All CPU capacities are equal.

i. `task_cpu(p)` is in the intersection of `cpus_mask` and `free_cpus`.

`find_later_rq()` returns `task_cpu(p)`.

ii. `task_cpu(p)` is not in the intersection of `cpus_mask` and `free_cpus`.

`find_later_rq()` returns a CPU that is both in the intersection and in the lowest-possible `sched_domain` that also contains `task_cpu(p)`. If there is no such CPU, an arbitrary CPU in the intersection is returned.

2. The intersection of `p->cpus_mask` and `rq->cpudl.free_cpus` is empty.

Then `find_later_rq()` considers the CPU at the root of `cpudl.elements`. Let this CPU be `best_cpu` with corresponding deadline `latest_deadline` in `elements`.

(a) `p` has affinity for (*i.e.*, `p->cpus_mask` includes) `best_cpu` and `p->dl.deadline` is earlier than `latest_deadline`.

`find_later_rq()` returns `best_cpu`.

(b) `p` does not have affinity for `best_cpu` or has a later deadline than `latest_deadline`.

`find_later_rq()` returns `-1`.

`push_dl_task()` does not call `find_later_rq()` directly, instead calling function `find_lock_later_rq()`, which is also `SCHED_DEADLINE`'s class-specific `find_lock_rq()` function (recall this `sched_class` function is used in `push_cpu_stop()`). Be aware of the distinction between `find_lock_rq()`, which is a function pointer in `sched_class`, and `find_later_rq()`, which is a helper function in `SCHED_DEADLINE`. `find_lock_later_rq()` shares the same prototype as `find_lock_rq()`, taking a task and runqueue pointers as arguments. The purpose of `find_lock_later_rq()` is to call `find_later_rq()` on said task, lock the runqueue corresponding to the CPU `find_later_rq()` returns, and return a pointer to the newly locked runqueue (or `NULL` if `find_later_rq()` returned `-1`).

`find_lock_later_rq()` may attempt to call `find_later_rq()` multiple times. Let the runqueue corresponding to the return value of `find_later_rq()` be the *tentative* runqueue. It is necessary to call `find_later_rq()` multiple times because in between `find_later_rq()` returning and acquiring

the tentative runqueue's spinlock, the state of the tentative runqueue may have changed such that the task being pushed would no longer be the earliest-deadline task on the tentative runqueue (*e.g.*, a task with an earlier deadline wakes on the tentative runqueue). After `find_later_rq()` returns, `find_lock_later_rq()` checks if such a state is observed, and if so, calls `find_later_rq()` again to attempt to push to a different runqueue. `find_later_rq()` can be called `DL_MAX_TRIES` (3) times before `find_lock_later_rq()` gives up and returns `NULL`.

In acquiring the tentative runqueue's spinlock, it may be necessary to unlock the pushing runqueue in order to acquire both runqueues' locks in CPU-index order. If `find_lock_later_rq()` observes that the state of the pushing runqueue or to-be-pushed task has changed while the pushing runqueue's spinlock was dropped (*e.g.*, the to-be-pushed task disabled migration, was migrated away by some other function, was scheduled, *etc.*), then `find_lock_later_rq()` immediately gives up and returns `NULL`.

Having covered `find_lock_later_rq()`, `push_dl_task()` is presented in Listing 4.23. At line 6, `push_dl_task()` calls `pick_next_pushable_dl_task()`, which returns the leftmost task in red-black tree `rq->dl.pushable_dl_tasks_root` to determine which task is to be pushed (*i.e.*, `next_task`). `pick_next_pushable_dl_task()` is similar in purpose to `pick_earliest_pushable_dl_task()`, used in `pull_dl_task()` to select a task to be pulled, except `pick_next_pushable_dl_task()` needs not consider the affinity mask of its returned task. At line 11, `push_dl_task()` gives up on pushing `next_task` if it has disabled migration. Otherwise, `push_dl_task()` calls `find_lock_later_rq()` to identify which runqueue to push `next_task` to.

In the case that `find_lock_later_rq()` returns `NULL`, then no suitable runqueue was found. In this case, `push_dl_task()` rechecks that `next_task` is the earliest-deadline pushable task on `rq` (recall that `find_lock_later_rq()` may temporarily release `rq`'s spinlock) by calling `pick_next_pushable_dl_task()` again. If `pick_next_pushable_dl_task()` returns `NULL` (*i.e.*, there are no longer pushable tasks on `rq`) or returns `next_task` (which cannot be pushed due to `find_lock_later_rq()` returning `NULL`), then `push_dl_task()` returns without pushing a task. Otherwise, `push_dl_task()` sets `next_task` to the new task and jumps back to line 11, attempting to push the new `next_task`.

On the other hand, if `find_lock_later_rq()` returns a runqueue pointer, then `next_task` is migrated to the later runqueue (lines 24-29).

```
1  int push_dl_task(struct rq *rq)
2  {
3      struct task_struct *next_task;
4      struct rq *later_rq = NULL;
5
6      next_task = pick_next_pushable_dl_task(rq);
7      if (!next_task)
8          return 0;
9
10     while (!later_rq) {
11         if (next_task->migration_disabled)
12             return 0;
13
14         later_rq = find_lock_later_rq(next_task, rq);
15         if (!later_rq) {
16             struct task_struct *task = pick_next_pushable_dl_task(rq);
17             if (!task || task == next_task)
18                 return 0;
19
20             next_task = task;
21         }
22     }
23
24     next_task->on_rq = TASK_ON_RQ_MIGRATING;
25     dequeue_task_dl(rq, next_task, 0);
26     migrate_task_rq_dl(next_task, later_rq->cpu);
27     __set_task_cpu(next_task, later_rq->cpu);
28     enqueue_task_dl(later_rq, next_task, ENQUEUE_MIGRATED);
29     next_task->on_rq = TASK_ON_RQ_QUEUED;
30
31     resched_curr(later_rq);
32     return 1;
33 }
```

Listing 4.23: Push task pseudocode.

```
1 int balance_dl(struct rq *rq, struct task_struct *p, struct rq_flags *rf)
2 {
3     if (!on_dl_rq(&p->dl) && dl_prio(p->prio))
4         pull_dl_task(rq);
5     return rq->dl.dl_nr_running > 0;
6 }
```

Listing 4.24: `balance_dl()`.

4.4.2.3 Suspending and Waking

Pushes and pulls are generally triggered by reschedules, which (ignoring CBS throttling and replenishments) are primarily caused by tasks suspending (a suspending task directly calls scheduling functions) and waking (a waking task may preempt the running task).

Suspending. A suspending task calls `__schedule()` to unschedule itself and remove itself from its runqueue. Recall Listing 4.4 of the `__schedule()` function. For `SCHED_DEADLINE`, a suspending task means that a task that previously had an early-enough deadline to be scheduled is no longer runnable. The CPU of the suspending task must choose a new task with an early deadline to schedule. This new task may need to be pulled from another runqueue. Pulling is done by `balance_dl()`, which will be called by `__schedule()` (recall the loop at line 16 of Listing 4.4).

`balance_dl()` is presented in Listing 4.24. `balance_dl()` checks if tasks need to be pulled to this `rq` by verifying that `p`, the previously scheduled task, is both a `SCHED_DEADLINE` task (`dl_prio(p->prio)`) and was suspended (not `on_dl_rq(&p->dl)`). `on_dl_rq()` is a function that returns `true` if a given `sched_dl_entity` is queued on any tree. These two conditions indicate that `p` was the earliest-deadline task on runqueue `rq` and has become unschedulable; thus, the other runqueues should be searched for a task with a potentially earlier deadline than the remaining tasks on runqueue `rq`, *i.e.*, `pull_dl_task()` must be called. `balance_dl()` returns whether there are `SCHED_DEADLINE` tasks on runqueue `rq`. Returning `true` indicates that the `balance()` functions of the lower scheduling classes need not be called.

This concludes discussion on how suspending results in tasks being pulled from other runqueues.

Waking. The waking function `try_to_wake_up()` causes two migrations. The first migration is of the waking task to the runqueue selected by `select_task_rq()`. This migration is performed without acquiring a runqueue lock, and has a primary purpose of enqueueing the waking task on *some* runqueue.

```
1  void select_task_rq_dl()(struct task_struct *p, int cpu, int flags)
2  {
3      int target;
4      struct dl_rq *dl_rq;
5
6      if (!(flags && WF_TTWU))return cpu;target = `find_later_rq(p);
7      if (target == -1)
8          return cpu;
9
10     dl_rq = &cpu_rq(target)->dl;
11     if (!dl_rq->dl_nr_running || p->dl.deadline < dl_rq->earliest_dl.curr)
12         return target;
13
14     return cpu;
15 }
```

Listing 4.25: `select_task_rq_dl()`.

Once the waking task has been enqueued on some runqueue, its priority is either higher or lower than the currently running task on said runqueue. The second migration is a push from this runqueue triggered within `task_woken()`. The purpose of this push is to migrate the waking task to a runqueue with only tasks with later deadlines. This push is performed with runqueue locks acquired.

We discuss the logic behind these migrations by stepping through the functions called by `try_to_wake_up()`. The first function call of note is to the class-specific `select_task_rq()` function, which returns the target runqueue of the aforementioned first migration. For `SCHED_DEADLINE`, this is `select_task_rq_dl()`, which is presented in Listing 4.25.

`select_task_rq_dl()` immediately returns if flag `WF_TTWU` is not set in `flags` (line 6). If `try_to_wake_up()` is unset, then `select_task_rq()` was called as a result of either a task forking (`WF_FORK`), which is not permitted in `SCHED_DEADLINE`, or replacing its binary (`WF_EXEC`), which does not necessitate a migration. For either `WF_FORK` or `WF_EXEC`, `select_task_rq_dl()` is called with argument `cpu` having the value of task `p`'s current CPU, which `select_task_rq_dl()` then immediately returns.

If `WF_TTWU` is set, then, as in `push_dl_task()`, `find_later_rq()` is used to identify the latest CPU to send waking task `p` to. Note that, unlike `push_dl_task()`, `select_task_rq_dl()` directly calls `find_later_rq()` instead of calling `find_lock_later_rq()`.

After calling `select_task_rq()`, `try_to_wake_up()` also potentially calls `migrate_task_rq()`. For `SCHED_DEADLINE`, `migrate_task_rq_dl()` updates statistics used by GRUB and is

```
void task_woken_dl(struct rq *rq, struct task_struct *p)
{
    struct task_struct *curr = rq->curr;
    if (curr != p &&
        dl_prio(curr->prio) &&
        curr->dl.deadline < p->dl.deadline)
        while (push_dl_task(rq));
}
```

Listing 4.26: `task_woken_dl()`.

independent of how tasks are migrated. `migrate_task_rq_dl()` will be discussed when discussing GRUB in Section 4.4.8.

Next, `try_to_wake_up()` enqueues the waking task by calling `enqueue_task()`. `enqueue_task_dl()` was discussed previously. If the waking task and the current task on the corresponding runqueue are of the same `sched_class`, `try_to_wake_up()` calls `wakeup_preempt()` to determine if the current task should be preempted. `wakeup_preempt_dl()` calls `resched_curr()` if the waking task has an earlier deadline `deadline` than the currently scheduled task.

The last `sched_class` function called by `try_to_wake_up()` is `task_woken()`. `task_woken_dl()` is presented in Listing 4.26. Having now enqueued the waking task `p` on the runqueue returned by `select_task_rq_dl()`, `task_woken_dl()` migrates `p` if this runqueue is no longer appropriate. This is the aforementioned second migration triggered by `try_to_wake_up()`. If `p` does not have an early enough deadline to preempt the current task on its runqueue, then `task_woken_dl()` calls `push_dl_task()` to migrate the task.

4.4.2.4 Other Scheduling Class Functions

There remain miscellaneous `sched_class` functions that are described here.

Putting, picking, and setting. We briefly discuss the remaining `SCHED_DEADLINE` `sched_class` functions called by `__schedule()`. These are `put_prev_task_dl()`, `pick_task_dl()`, and `set_next_task_dl()`.

After balancing, `__schedule()` calls the `put_prev_task()` function on the task being unscheduled. For `SCHED_DEADLINE`, this is `put_prev_task_dl()` (Listing 4.27). The primary function of `put_prev_task_dl()` is to potentially call `enqueue_pushable_dl_task()` on `p`. This is neces-

```
1  void put_prev_task_dl(struct rq *rq, struct task_struct *p)
2  {
3      update_curr_dl(rq);
4      if (on_dl_rq(&p->dl) && p->nr_cpus_allowed > 1)
5          enqueue_pushable_dl_task(rq, p);
6  }
```

Listing 4.27: `put_prev_task_dl()`.

```
1  void set_next_task_dl(struct rq *rq, struct task_struct *p, bool first)
2  {
3      p->se.exec_start = rq_clock_task(rq);
4
5      dequeue_pushable_dl_task(rq, p);
6
7      if (!first)
8          return;
9
10     hrtick_start(rq, p->dl.runtime);
11
12     deadline_queue_push_tasks(rq);
13 }
```

Listing 4.28: `set_next_task_dl()`.

sary because `p`, having been unscheduled, must be made available to be pushed to or pulled by other CPUs. The call to `update_curr_dl()` will be explained later when CBS throttling is discussed.

Next in `__schedule()` are calls to the `pick_task()` functions. `pick_task_dl()` is trivial. The leftmost (*i.e.*, earliest-deadline) task on `root` is returned, or `NULL` if no `SCHED_DEADLINE` tasks are enqueued on this runqueue.

If `pick_task_dl()` returns a task in `__schedule()`, `set_next_task_dl()` (Listing 4.28) is called on this task to denote it as the scheduled task. The setting of the `exec_start` field (line 3), which stores when task `p` was initially scheduled and the call to `hrtick_start()` at line 10 will be explained later when discussing CBS throttling (Section 4.4.3).

Inversely from `put_prev_task_dl()`, `set_next_task_dl()` calls `dequeue_pushable_dl_task()` due to task `p` being scheduled (and thus, no longer migratable).

`set_next_task_dl()` ends by calling function `deadline_queue_push_tasks()`, which causes `push_dl_task()` to be called on runqueue `rq` after `p` has become the scheduled task. `deadline_queue_push_tasks()` does this by queueing `push_dl_task()` onto the queue of callback functions

rq->balance_callback. The purpose of queueing push_dl_task() here is to migrate the task preempted by p to a later CPU.

Note that, when called from __schedule(), argument first, which indicates that set_next_task() was not called from the change pattern, is true. Thus, set_next_task_dl() does not return at line 8.

Entering and leaving. Policy changes call the sched_class functions switched_to(), switched_from(), and prio_changed(). For SCHED_DEADLINE, these function pointers point to switched_to_dl(), switched_from_dl(), and prio_changed_dl(). Be aware that these functions *do not* contain ACS code. The purpose of these functions is to migrate tasks in response to tasks entering and leaving SCHED_DEADLINE. For example, switched_from_dl(), called when a task leaves SCHED_DEADLINE, calls pull_dl_task(). This should be intuitive because, from the point of view of SCHED_DEADLINE, a task leaving SCHED_DEADLINE is analogous to said task permanently suspending. Recall that a suspending task calls pull_dl_task() via calling balance_dl() in __schedule().

Runqueue onlining and offlining. sched_class functions rq_online() and rq_offline() are called whenever a CPU is added to a root_domain. For SCHED_DEADLINE, rq_online_dl() and rq_offline_dl() are responsible for initializing and clearing the SCHED_DEADLINE-relevant state for the corresponding CPU in the root_domain of interest. This state is the CPU's corresponding bit in mask dlo_mask as well as its presence in either cpudl's heap elements or mask free_cpus.

4.4.3 CBS

Linux tasks do not naturally have deadlines because they are not obligated to follow any real-time task model. The deadline of a task arises from its encapsulating CBS, which increments the deadline whenever the task exhausts its budget. The CBS also throttles tasks that exhaust their budget.

clock vs. clock_task. There are several points in the CBS logic where it is necessary to know the current time instant (*e.g.*, when setting the first deadline of a task when it enters SCHED_DEADLINE or when computing how much budget to decrease for a given interval of execution). Recall that the current time instant is stored in rq members clock, or its derived quantities clock_task and clock_pelt. For a given runqueue rq, if rq->clock is a measure of the local (rq's CPU's) time, then rq->clock_

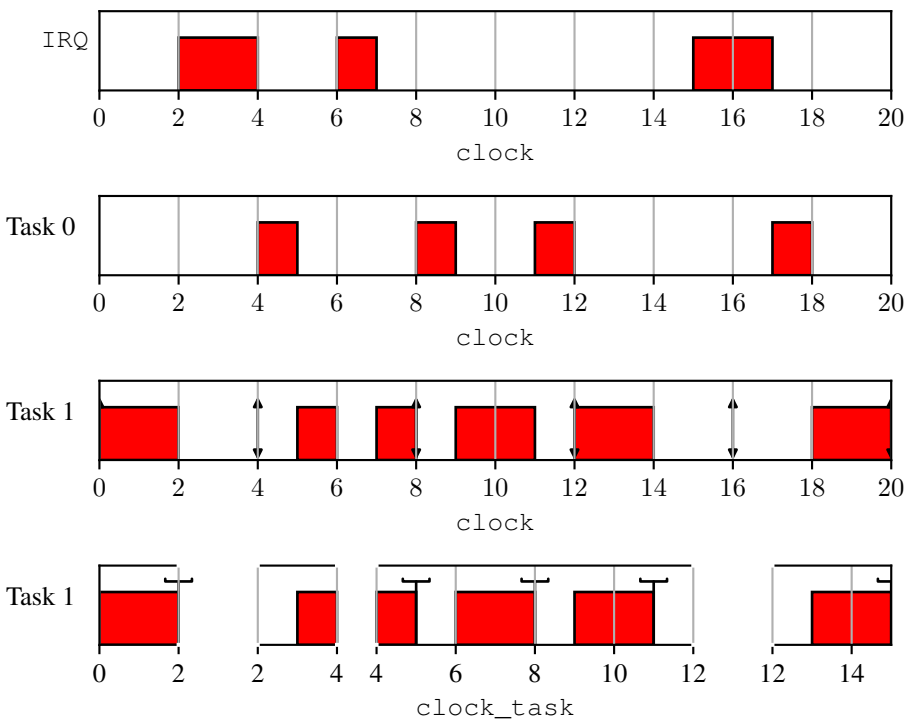


Figure 4.8: `clock_task` example.

`task` can be interpreted as a measure of time such that time spent executing interrupts does not contribute to `clock_task`, *i.e.*, time only advances while executing tasks or idling.

▼ **Example 4.6.** Consider the schedule in Figure 4.8. The first row illustrates the execution of IRQs, the second row illustrates the execution of a task with PID 0, and the third and fourth row illustrate the execution of a task with PID 1. The first through third row share a horizontal time axis as would be observed by `clock`, while the fourth row has a time axis as observed by `clock_task`. Observe that `clock_task` does not does not advance whenever an IRQ is executing.

Suppose we want to decrement Task 1's budget for execution within the time interval represented by $[5, 8)$ in the `clock` axis and $[3, 5)$ in the `clock_task` axis. The scheduler, which is only informed when *tasks* are scheduled, is only aware that Task 1 was the only task scheduled in this interval, and is oblivious to time spent executing IRQs. As such, if the scheduler were to observe time according to the `clock` axis, it would decrement $8 - 5 = 3$ units of budget. In comparison, if the scheduler were to observe time according to `clock_task`, it would decrement $5 - 3 = 2$ units of budget. This reflects the actual amount of execution provided to Task 1. ▲

The CBS refers to `clock` when setting deadlines and replenishment times and to `clock_task` when doing budget calculations.

Throttling. CBS throttling occurs when tasks exhaust their budgets. Idealized budget-based servers decrease their budgets continuously with time. This is not possible on a real system. Instead, `SCHED_DEADLINE` maintains an invariant: the actual budget of any unscheduled task must match its ideal budget, and the actual budget of any scheduled task must have a budget update queued on an `hrtimer` set to fire no later than (within a small margin of error) the exhaustion time of its ideal budget. Maintaining this invariant allows `SCHED_DEADLINE` to correctly throttle tasks at ideal budget exhaustion times.

CBS budget updates occur on calls to `update_curr_dl()`, which is the `update_curr()` function of `SCHED_DEADLINE`. We will discuss specifics of `update_curr_dl()` after discussing how and when it is called. That budgets of unscheduled tasks match ideal budgets is maintained by calling `update_curr_dl()` in `put_prev_task_dl()` (line 3 of Listing 4.27). `put_prev_task_dl()` is called whenever a task is unscheduled (line 20 of Listing 4.4).

Maintaining the invariant for scheduled tasks is more complicated. `update_curr_dl()` is called from `task_tick_dl()`, `SCHED_DEADLINE`'s `task_tick()` function. Recall from Section 4.3.3 that `task_tick()` is called from `hrtick()`, the timer function of the per-runqueue `hrtimers` `hrtick_timer`. To maintain the invariant for a scheduled task, `hrtick_timer` on said task's runqueue is armed to fire at the expected time instant that budget would be depleted, *i.e.*, the current time added to the task's remaining budget. Firing at this time instant causes `hrtick()` to call `task_tick_dl()`, which calls `update_curr_dl()`, which updates the budget.

Maintaining that `hrtick_timer` is always armed at the budget exhaustion time of the scheduled task involves the functions `set_next_task_dl()`, `task_tick_dl()`, and `__schedule()`. `set_next_task_dl()`, called whenever a task is scheduled (line 25 of Listing 4.4), arms `hrtick_timer` based on the remaining budget, *i.e.*, `runtime` of the corresponding `sched_dl_entity` (line 10 of Listing 4.28). While at first glance, this is sufficient to maintain the invariant, the task may consume less than `runtime` units of budget over `runtime` time units of execution. Less than `runtime` units may be consumed under GRUB, asymmetric capacities, and DVFS, which will be detailed in Sections 4.4.8-4.4.9. Note that budget for tasks is relative to execution on a maximum capacity CPU operating at maximum frequency, thus, more than `runtime` units of budget cannot be consumed in `runtime` time units.

```
void task_tick_dl(struct rq *rq, struct task_struct *p, int queued)
{
    update_curr_dl(rq);
    if (queued && p->dl.runtime > 0 && is_leftmost(p, &rq->dl))
        hrtick_start(rq, p->'dl.runtime);
}
```

Listing 4.29: `task_tick_dl()`.

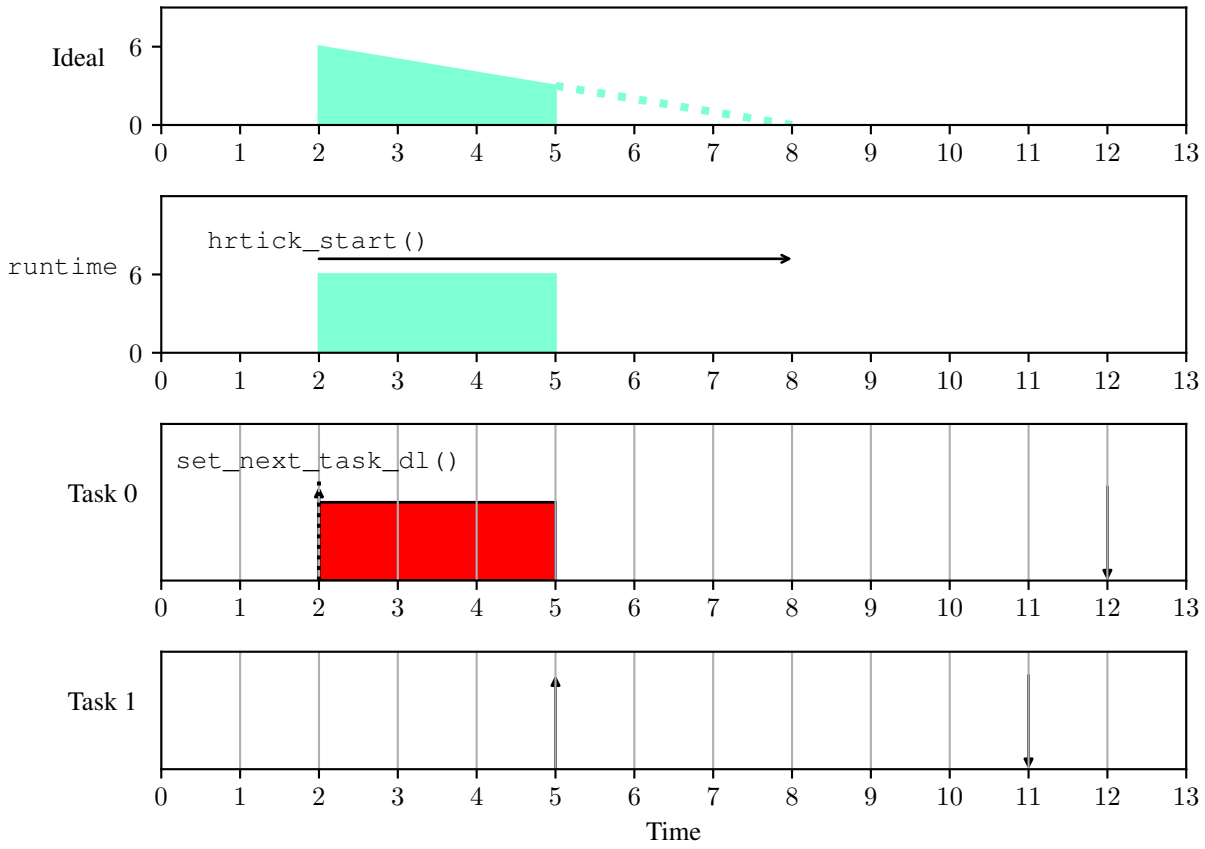
To maintain the invariant for scheduled tasks, `task_tick_dl()` rearms `hrtick_timer` when it observes that budget has not been exhausted. Consider Listing 4.29. `task_tick_dl()` first calls `update_curr_dl()` to update the budget of the running task. `hrtick_timer` is then rearmed if `task_tick_dl()` was called by `hrtick_timer` firing (*i.e.*, `queued` is set), budget is not exhausted (`p->dl.runtime` > 0), and the current task is still the task with the earliest deadline on the runqueue (`is_leftmost(p, &rq->dl)`). `is_leftmost()` returns `true` if `p` is the leftmost (*i.e.*, has the earliest deadline) task in tree `rq->dl.root`. `update_curr_dl()` may update the current task `p`'s deadline (this will be discussed later when covering CBS budget replenishment), thereby changing its position in this tree. If `p` is no longer the leftmost task, then it will soon be unscheduled, and there is no need to arm `hrtick_timer`.

`__schedule()` cancels `hrtick_timer` (line 9 of Listing 4.4). There is no need to keep `hrtick_timer` armed for a task when `__schedule()` is called, as `put_prev_task_dl()` will call `update_curr_dl()`. Canceling `hrtick_timer` also prevents `task_tick()` from hitting a task other than the task that was scheduled when `hrtick_timer` was armed.

The following example demonstrates how `runtime` is decremented to match an ideal CBS budget.

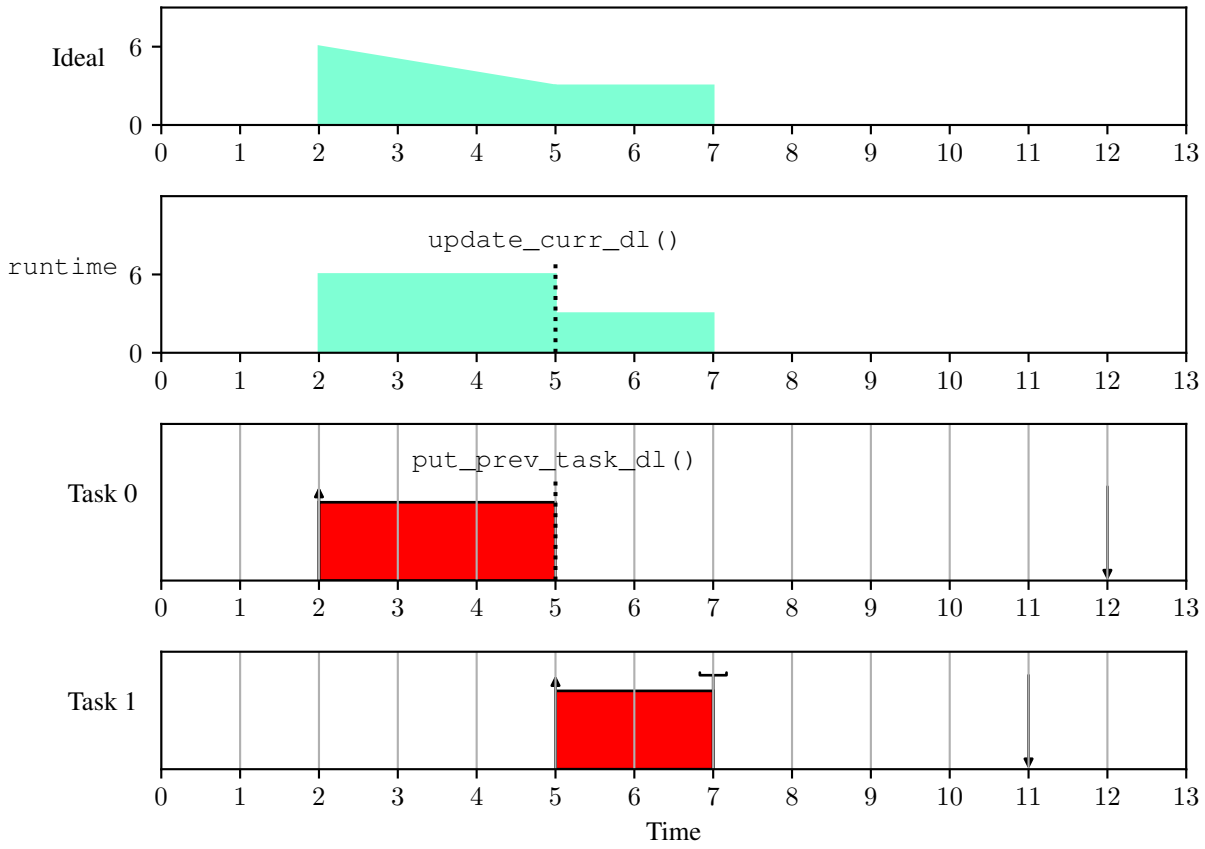
▼ **Example 4.7.** Consider Task 0 with `dl_runtime` of 6 and `dl_deadline` of 10 and Task 1 with `dl_runtime` of 2 and `dl_deadline` of 6 running on a single CPU. Task 0 starts at time 2 and Task 1 starts at time 5. Figure 4.9 illustrates the idealized and actual budget (*i.e.*, `runtime`) of Task 0.

The schedule of this system over interval $[2, 5)$ is illustrated in Figure 4.9a. Both the ideal budget and `runtime` start with 6 units at time 2. While the ideal budget decreases continuously with time over $[2, 5)$, `runtime` is constant. Instead, `set_next_task_dl()`, called on Task 0 when it was scheduled at time 2, calls `hrtick_start()` to arm `hrtick_timer` on the CPU's runqueue to fire `runtime` (6) time units after the current time (2). Observe that this firing time is time 8, the expected exhaustion time of Task 0's ideal budget.



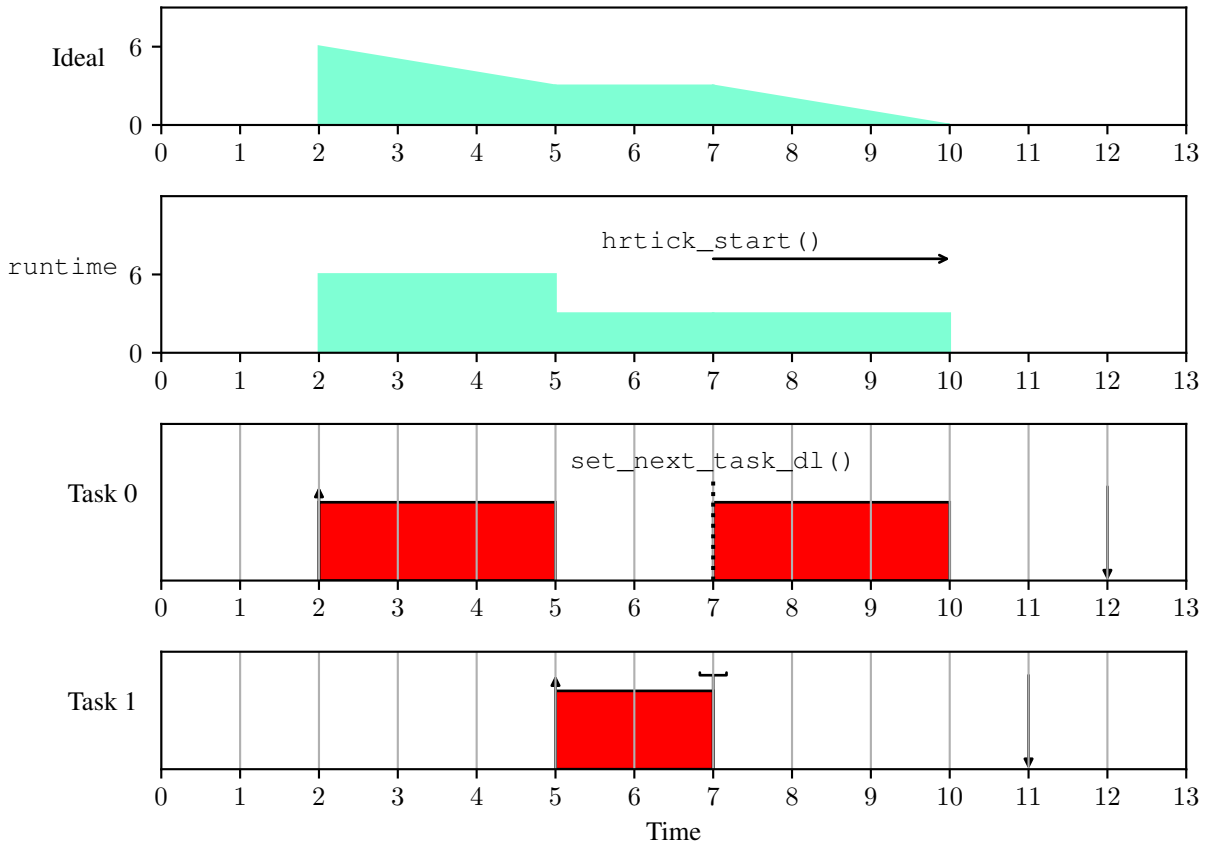
(a) Task 0 scheduled over $[2, 5)$.

Figure 4.9: `runtime` vs. ideal budget.



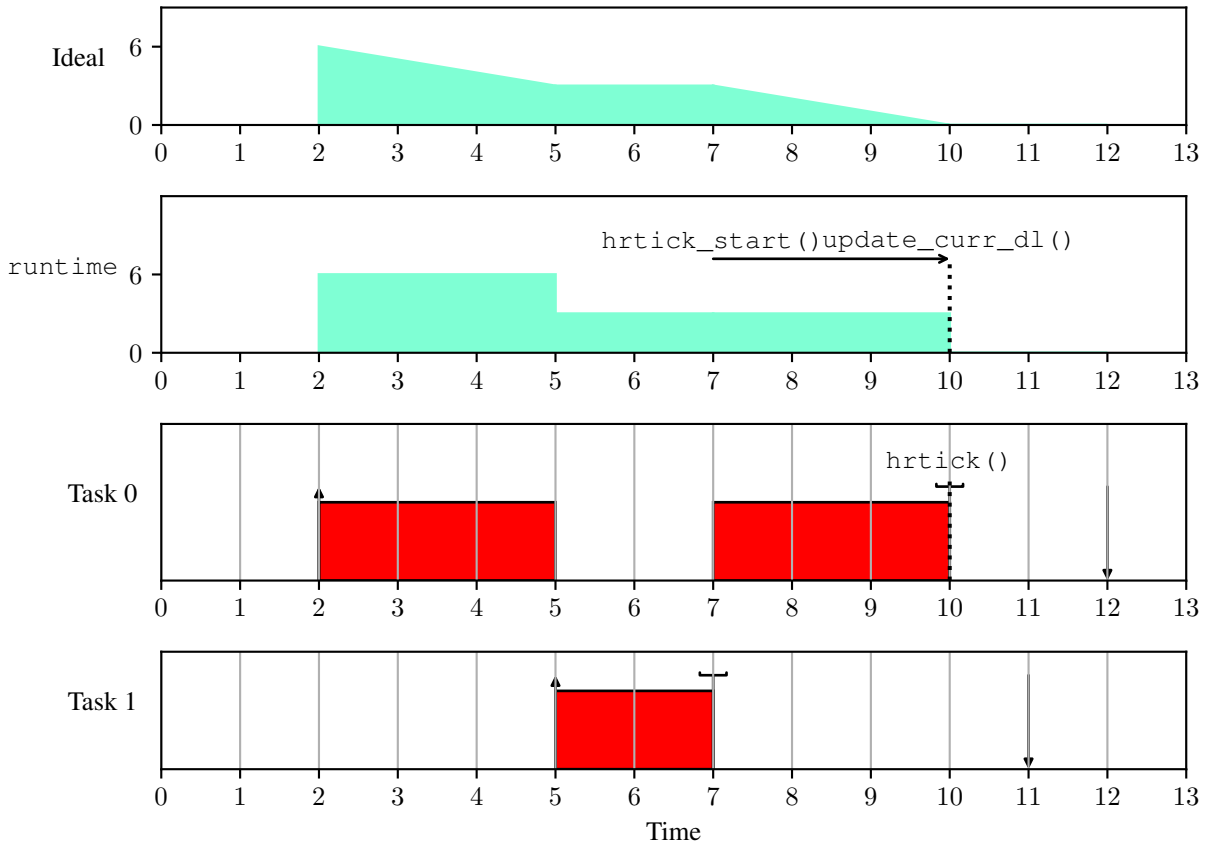
(b) Task 0 unscheduled over $[5, 7)$.

Figure 4.9: `runtime` vs. ideal budget (continued).



(c) Task 0 scheduled over $[7, 10)$.

Figure 4.9: `runtime` vs. ideal budget (continued).



(d) Task 0 throttled at time 10 by `hrtick()`.

Figure 4.9: `runtime` vs. ideal budget (continued).

```
void yield_task_dl()(struct rq *rq)
{
    rq->curr->dl.dl_yielded = 1;
    update_curr_dl(rq);
}
```

Listing 4.30: `yield_task_dl()`.

At time 5, Task 0 is preempted by Task 1 (Figure 4.9b). `hrtick_timer` is canceled in `__schedule()`, removing the firing at time 8 illustrated in Figure 4.9a. When Task 0 is unscheduled, `put_prev_task_dl()` calls `update_curr_dl()` to set `runtime` to the same value as the ideal budget (3). While Task 0 is unscheduled over $[5, 7)$, neither the ideal budget nor `runtime` change from 3 units.

At time 7, Task 0 is again scheduled (Figure 4.9c). Upon being scheduled, `set_next_task_dl()` is called on Task 0, which again calls `hrtick_start()` to arm `hrtick_timer` to fire `runtime` (3) time units after the current time (7). This is time 10, which, because Task 0 is not preempted in interval $[7, 10)$, is the actual exhaustion time of the ideal budget.

At time 10, `hrtick_timer` fires and calls `hrtick()`, which calls `task_tick_dl()`, which calls `update_curr_dl()` to set `runtime` to the value of the ideal budget (0). `update_curr_dl()` observes that Task 0 has depleted `runtime`, and throttles Task 0 until its replenishment at its next period. ▲

The last location where `update_curr_dl()` is called is in `yield_task_dl()`. `yield_task_dl()` (Listing 4.30) invokes the current task to immediately throttle itself. This is done by setting the `dl_yielded` flag in the current task's `sched_dl_entity` and calling `update_curr_dl()`. `update_curr_dl()` observes this flag and behaves as if `runtime` was depleted.

Having discussed how `update_curr_dl()` is called, we now discuss its implementation. Before discussing pseudocode, we discuss `exec_start`, a member of the EEVDF scheduling entity used by `update_curr_dl()`. As stated previously, `update_curr_dl()` sets `runtime` to the ideal budget's value. This is done by decreasing `runtime` by the units of execution completed since the later of when `runtime` was last updated (*i.e.*, `update_curr_dl()` was called) or the current task was scheduled (*i.e.*, `set_next_task_dl()` was called). The scheduler happens to store the latest time instant that either

```
1  void update_curr_dl(struct rq *rq)
2  {
3      struct task_struct *curr = rq->curr;
4      struct sched_dl_entity *dl_se = &curr->dl;
5      u64 delta_exec, scaled_delta_exec now;
6      int cpu = rq->cpu;
7
8      now = rq_clock_task(rq);
9      delta_exec = now - curr->se.exec_start;
10     curr->se.exec_start = now;
11
12     /* delta_exec to scaled_delta_exec by GRUB or Asym. Cap. & DVFS */
13
14     dl_se->runtime -= scaled_delta_exec;
15
16 throttle:
17     if (dl_se->runtime <= 0 || dl_se->dl_yielded) {
18         dl_se->dl_throttled = 1;
19
20         dequeue_task_dl(rq, curr, 0);
21         if (dl_se->pi_se != dl_se || !start_dl_timer(curr))
22             enqueue_task_dl(rq, curr, ENQUEUE_REPLENISH);
23
24         if (!is_leftmost(curr, &rq->dl))
25             resched_curr(rq);
26     }
27 }
```

Listing 4.31: `update_curr_dl()`.

`update_curr()` or `set_next_task()` was called for any scheduling class except `idle_sched_class`. This time instant is stored in `exec_start`, *i.e.*, for task `p`, in `p->se.exec_start`.

Pseudocode for `update_curr_dl()` is presented in Listing 4.31. In lines 8-10, `update_curr_dl()` sets `delta_exec` to the duration of time spent executing the task since the previous `update_curr_dl()` call. This duration is the difference between the current time `now` and `exec_start`, which is set in `update_curr()` (line 10 in Listing 4.31) and `set_next_task()` (line 3 in Listing 4.28). `exec_start` is set to `now`'s value in `update_curr()`. Note that `exec_start` and `now` are always set by `rq_clock_task()`. Thus, `delta_exec`, the difference of these two values, does not include time spent executing IRQs.

`delta_exec` is then scaled to `scaled_delta_exec` by either GRUB or by the executing CPU's capacity and frequency. How `scaled_delta_exec` is computed will be discussed in Sections 4.4.8

(GRUB) and 4.4.9 (capacities and DVFS). `scaled_delta_exec` represents the actual amount of execution that `runtime` is decremented by (line 14).

After computing the new `runtime` value, `update_curr_dl()` checks if the current task should be throttled, *i.e.*, `runtime` has been exhausted or `dl_yielded` has been set (line 17), until the next replenishment time. If the task should be throttled, then it is dequeued.

A task dequeued due to throttling may be immediately re-enqueued if the next replenishment time has already passed or the task is inheriting another task's priority (line 21). This enqueue is called with flag `ENQUEUE_REPLENISH`, which adds maximum budget `dl_runtime` to the current budget `runtime` (`enqueue_task_dl()` will be discussed in later paragraphs). We call this immediate enqueueing of the task a *bypassed* throttle.

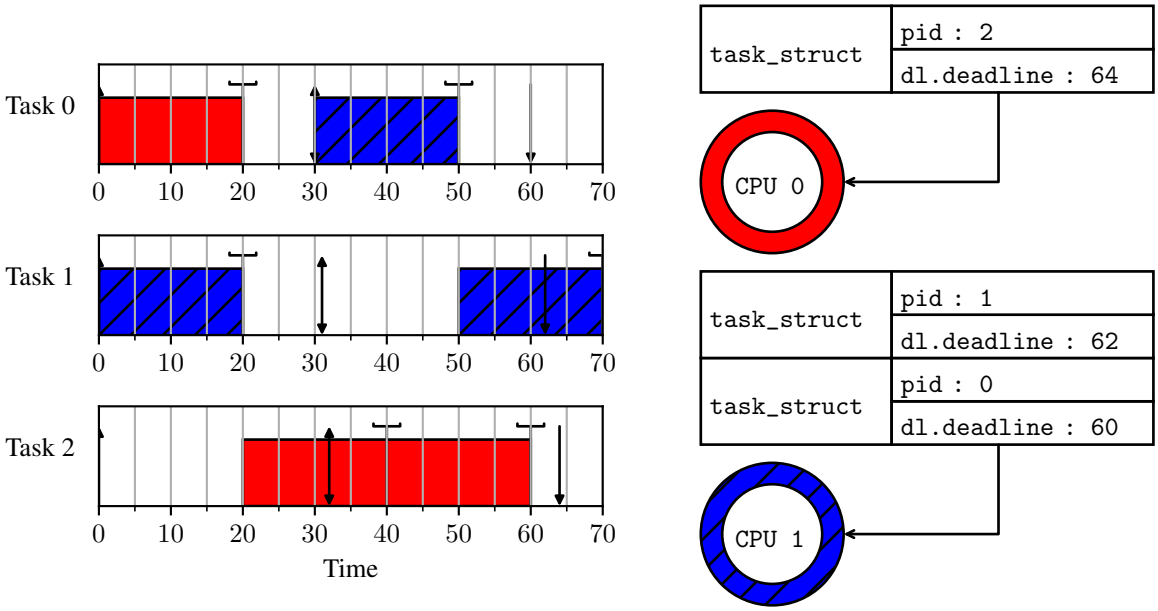
`update_curr_dl()` determines if the next replenishment time has passed by checking whether or not `start_dl_timer()` returns 0. `start_dl_timer()` will be discussed in later paragraphs.

Whether the task is inheriting priority is determined by checking whether the task's `pi_se` pointer points to itself. Recall that this pointer is set in function `rt_mutex_setprio()` (line 23 of Listing 4.10) to point to the `sched_dl_entity` of the task being inherited from. Why priority-inheriting tasks bypass throttles will be discussed in Section 4.4.7.

At line 24, `update_curr_dl()` determines if rescheduling is necessary due to the current task being throttled by checking if the current task is not leftmost in the deadline-ordered tree `root`. If throttling was bypassed, as part of being enqueued with `ENQUEUE_REPLENISH`, the deadline of the current task may have increased, making the current task no longer the leftmost task in `root`. Then the new leftmost task has an earlier deadline and should be scheduled. If throttling was not bypassed, then the current task can no longer be scheduled. Then the current task is not leftmost in `root` because it is removed from this tree when dequeued.

Bypassed throttle bug. There appears to be a defect in `update_curr_dl()` and `balance_dl()` when throttling is bypassed that causes `SCHED_DEADLINE` to act differently from EDF. This is illustrated in the following example.

▼ **Example 4.8.** Consider three implicit-deadline tasks with `dl_runtime` of 20. Let Task 0 have `dl_period` of 30, Task 1 have `dl_period` of 31, and Task 2 have `dl_period` of 32. These three



(a) Task 2 bypasses throttling at time 40.

(b) Runqueues at time 40.

Figure 4.10: Out-of-deadline-order execution at time 40.

tasks execute on two CPUs and begin executing at time 0. A schedule of this system is illustrated in Figure 4.10a.

At time 30, Task 0 is replenished. At this time, Task 2, with `deadline` of 32, has an earlier deadline than Task 0, with `deadline` of 60. Thus, Task 0 is pushed from its original CPU, CPU 0, to CPU 1 and begins executing.

At time 31, Task 1 is replenished. With `deadline` of 62, it cannot preempt either of Task 0 or Task 2, and thus remains unscheduled on CPU 1's runqueue.

At time 40, Task 2 depletes its `runtime`. However, because its next replenishment time, 32, is in the past, Task 2 bypasses throttling in `update_curr_dl()`. Thus, Task 2 is immediately re-enqueued on CPU 0's runqueue. The state of the runqueues at this time is shown in Figure 4.10b. Because Task 2 remains the earliest-deadline task on CPU 0's runqueue, `update_curr_dl()` does not call `resched_curr()` (see line 24 of Listing 4.31). Task 2 remains scheduled despite unscheduled task Task 1 having an earlier deadline. ▲

The problem in `update_curr_dl()` that causes the above example is that the check on line 24 of Listing 4.31 does not consider the possibility that tasks with earlier deadlines may be on other runqueues.

The other runqueues are only observed during a pull in `balance_dl()`, which is only called as part of a reschedule.

Note that unconditionally calling `resched_curr()` in `update_curr_dl()` is insufficient to fix this defect. `balance_dl()` will not call `pull_dl_task()` if the previously scheduled task is still on the runqueue, which is the case for a task that has bypassed throttling.

Replenishment from throttling. The topic of replenishment can be broadly divided into how replenishment is armed via timers and how replenishment is implemented in `enqueue_task_dl()`'s CBS logic. We cover how replenishment is armed first.

If throttling is not bypassed, then the task must be replenished (*i.e.*, enqueued with `dl_runtime` added to `runtime`) at a later time. This later time is the next replenishment time, which is computed as the period `dl_period` added to the difference between the absolute deadline `deadline` and relative deadline `dl_deadline`. Note how, for a sporadic task, this computes the next arrival time, which is analogous to the replenishment time under CBS.

In `update_curr_dl()`, function `start_dl_timer()` arms `hrtimer dl_timer` in the task's `sched_dl_entity` to fire at the next replenishment time (assuming it has not already passed). `dl_timer`'s callback function is `dl_task_timer()` (Listing 4.32). At the next replenishment time, `dl_task_timer()` enqueues and replenishes the `runtime` of its throttled task (line 9). After `runtime` is replenished and the task is enqueued, the scheduling and migration logic in `dl_task_timer()` (lines 11-16) is similar to that of `try_to_wake_up()` (lines 20-26 of Listing 4.5). This should be expected because in both `dl_task_timer()` and `try_to_wake_up()`, a task is becoming runnable.

Having discussed how `enqueue_task_dl()` is called in the timer callback function `dl_task_timer()`, we now discuss the internals of `enqueue_task_dl()`'s CBS logic (Listing 4.33). This CBS logic examines the `flags` argument to determine the context `enqueue_task_dl()` was called from. Being called with `ENQUEUE_REPLENISH` signifies the task is being enqueued due to a task being unthrottled (which may occur in `dl_task_timer()`, the throttle being bypassed in `update_curr_dl()`, or the task inheriting priority in `rt_mutex_setprio()`).

On observing `ENQUEUE_REPLENISH`, `enqueue_task_dl()` calls `replenish_dl_entity()`. Function `replenish_dl_entity()` (Listing 4.34) increments the task's budget `runtime` and absolute deadline `deadline`.

```
1  enum HRTIMER_RESTART dl_task_timer(struct hrtimer *timer)
2  {
3      struct sched_dl_entity *dl_se = container_of(timer,
4                                                   struct sched_dl_entity,
5                                                   dl_timer);
6      struct task_struct *p = container_of(dl_se, struct task_struct, dl);
7      struct rq *rq = task_rq(p);
8
9      enqueue_task_dl(rq, p, ENQUEUE_REPLENISH);
10
11     if (dl_prio(rq->curr->prio))
12         wakeup_preempt_dl(rq, p, 0);
13     else
14         resched_curr(rq);
15
16     push_dl_task(rq);
17
18     return HRTIMER_NORESTART;
19 }
```

Listing 4.32: `dl_task_timer()`.

```
1  void enqueue_task_dl(struct rq *rq, struct task_struct *p, int flags)
2  {
3      ⋮
4
5      if (flags & ENQUEUE_WAKEUP)
6          update_dl_entity(p->dl);
7      else if (flags & ENQUEUE_REPLENISH)
8          replenish_dl_entity(p->dl);
9      else if (flags & ENQUEUE_RESTORE)
10         if (p->dl.deadline < rq_clock(task_rq(p)))
11             setup_new_dl_entity(p->dl);
12
13     ⋮
14 }
```

Listing 4.33: `enqueue_task_dl()` CBS logic.

```
void replenish_dl_entity(struct sched_dl_entity *dl_se)
{
    while (dl_se->runtime <= 0) {
        dl_se->deadline += dl_se->pi_se->dl_period;
        dl_se->runtime += dl_se->pi_se->dl_runtime;
    }

    dl_se->dl_yielded = 0;
    dl_se->dl_throttled = 0;
}
```

Listing 4.34: `replenish_dl_entity()`.

Wakeup rules. Linux's CBS logic defines conditions under which, on wakeup, current budget `runtime` and absolute deadline `deadline` are preserved, reset, or scaled. By *preserved*, we mean that on wakeup, a task's `runtime` and `deadline` values are unchanged from when the task suspended. By *reset*, we mean that on wakeup, `runtime` is set to `dl_runtime` and `deadline` is set to `dl_deadline` added to the wakeup time. By *scaled*, we mean that on wakeup, a task's `runtime` is set to a value proportional to the duration of time until `deadline`. `deadline` is unchanged. These rules are designed such that the CBS satisfies certain HRT properties (Buttazzo and Abeni, 1998; Abeni et al., 2015), so we do not justify them in this dissertation. Whether `runtime` and `deadline` are preserved, reset, or scaled on wakeup is determined in the `update_dl_entity()` function called by `enqueue_task_dl()` (line 6 of Listing 4.33).

`runtime` and `deadline` are preserved if `deadline` is at least the wakeup time and

$$\frac{\texttt{runtime}}{\texttt{deadline} - \texttt{wakeup time}} \leq \frac{\texttt{pi\_se->dl\_runtime}}{\texttt{pi\_se->dl\_deadline}}. \tag{4.3}$$

(4.3) being true indicates that the remaining budget `runtime` can be conceptually treated as its own separate job with deadline `deadline` without exceeding the task's density.

Otherwise, the wakeup time occurred after `deadline` or (4.3) is false. For implicit-deadline tasks, `runtime` and `deadline` are always reset. For tasks such that `dl_deadline` is less than `dl_period`, if the wakeup time occurred before `deadline`, then `runtime` is scaled. Specifically, `runtime` is set to the product `dl_density` and the difference between `deadline` and the wakeup time. Otherwise, if the wakeup time occurred after `deadline`, then `runtime` and `deadline` are reset.

Change pattern. The remaining case to be discussed in Listing 4.33 is that `ENQUEUE_RESTORE` is set in `flags`. `ENQUEUE_RESTORE` indicates that `enqueue_task_dl()` was called due to the change pattern. The most important case of this flag is when the change pattern is used to change a task's policy to `SCHED_DEADLINE`. Function `setup_new_dl_entity()`, called by `enqueue_task_dl()`, sets the initial values of the new `SCHED_DEADLINE` task. `runtime` and `deadline` are set similarly to a reset under a wakeup, only with the wakeup time replaced with the policy request time.

`enqueue_task_dl()` may be called with `ENQUEUE_RESTORE` due to other uses of the change pattern such as affinity change requests. For such changes, `runtime` and `deadline` should not be reset. `enqueue_task_dl()` differentiates these other changes from policy change requests by checking if `deadline` is prior to the current time (line 10 of Listing 4.33). This is because the default value of

deadline for non-SCHED_DEADLINE tasks is 0. Thus, a non-SCHED_DEADLINE task will always pass the check on line 10.

Note that under multiprocessor scheduling, it is possible for the current time to exceed deadline, *i.e.*, a task is tardy. A tardy task will also pass the check on line 10. Thus, requests on behalf of a tardy task that invoke the change pattern such as changing affinities or locking mutexes will have the unintended side effect of resetting the task's runtime and deadline.

4.4.4 Admission Control

Recall from the discussion in Sections 4.1.5 and 4.4.1 that the ACS enforces that (4.2), which re-expresses (4.1) in terms of Linux variables, is true for every root_domain. (4.2) must be re-checked whenever a task enters SCHED_DEADLINE, a SCHED_DEADLINE task changes its parameters, a task is added to or removed from a root_domain, a root_domain changes its CPUs, or the system parameters sched_rt_runtime_us and sched_rt_period_us are modified.

Different functions check (4.2) depending on what change in the system resulted in the re-check. For policy changes to SCHED_DEADLINE or changing of existing SCHED_DEADLINE parameters, these checks are done in __sched_setscheduler(). A task changing its root_domain is checked in cpuset_can_attach(). Modifying the CPUs in a root_domain is checked in dl_cpuset_cpumask_can_shrink(). Changes to sched_rt_runtime_us and sched_rt_period_us are checked in sched_dl_global_validate(). Outside of sched_dl_global_validate(), these functions all call the same helper function __dl_overflow(). __dl_overflow() has the following prototype.

```
bool __dl_overflow(struct dl_
    bw *dl_b, unsigned long cap, u64 old_bw, u64 new_bw);
```

dl_b is the dl_bw of the root_domain being checked. cap is the total capacity of CPUs in the root_domain. old_bw is the total bandwidth of any tasks requesting SCHED_DEADLINE parameter changes, and new_bw is the prospective total bandwidth of these tasks if these changes are accepted. __dl_overflow() returns true if the ACS is enabled and (4.2) would be violated.

Affinities. Besides ensuring that (4.2) is maintained, the ACS also restricts tasks' affinities such that each task must have affinity for every CPU in its root_domain. This must be checked when tasks enter SCHED_DEADLINE via __sched_setscheduler() and when tasks set their affinities via __sched_setaffinity(). Pseudocode for these checks is presented in Listing 4.35.

```
1  int __sched_setscheduler()(struct task_struct *p, struct sched_attr *attr)
2  {
3          ⋮
4      if (policy == SCHED_DEADLINE && !__checkparam_dl(attr))
5          return -EINVAL;
6      if (dl_bandwidth_enabled() && policy == SCHED_DEADLINE) {
7          cpumask_t *span = rq->rd->span;
8          if (!cpumask_subset(span, p->cpus_ptr))
9              return -EPERM;
10     }
11     if ((policy == SCHED_DEADLINE || dl_prio(p->prio)) && sched_dl_
           overflow(p, policy, attr))
12         return -EBUSY;
13         ⋮
14 }
15
16 int __sched_setaffinity(struct task_struct *p, struct cpumask *mask)
17 {
18         ⋮
19     if (p->policy == SCHED_DEADLINE && dl_bandwidth_enabled() && !cpumask_
           subset(task_rq(p)->rd->span, new_mask))
20         return -EBUSY;
21         ⋮
22 }
```

Listing 4.35: ACS with affinities.

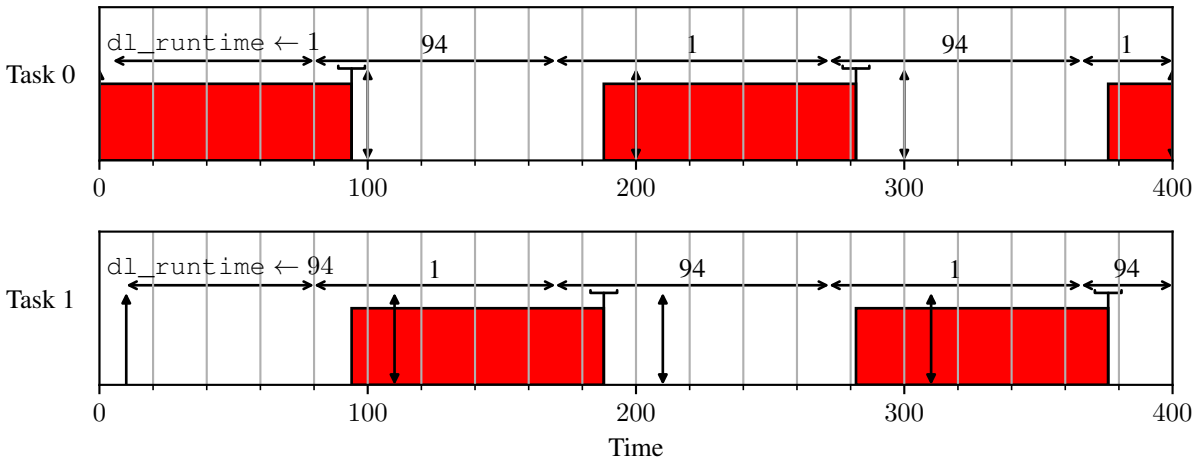


Figure 4.11: Unbounded response times due to dynamic tasks

We start with `__sched_setscheduler()`. At line 4, `__sched_setscheduler()` calls `__checkparam_dl()`, which verifies that the requested relative deadline of a task attempting to change its policy to `SCHED_DEADLINE` (or a `SCHED_DEADLINE` task changing its parameters) is at most its requested period. Technically, this check does not fall under the ACS as the check is performed regardless of whether the ACS is enabled or not (recall from Section 4.1.5 that the ACS can be disabled by writing to `sched_rt_runtime_us`).

The check in lines 6-8 verifies that the task has affinity for each CPU in the `root_domain`. `cpumask_subset()` if the former CPU bitmask is a subset of the latter. The function `dl_bandwidth_enabled()` returns `true` when the ACS is enabled. As such, this check is ignored when the ACS is disabled.

The check at line 11 checks that the requested parameters would not violate (4.2) if enacted. Function `sched_dl_overflow()` is a wrapper of `__dl_overflow()`, discussed earlier in Section 4.4.4.

In `__sched_setaffinity()`, the check at line 19 is the same check performed over lines 6-8 in `__sched_setscheduler()`.

Instantaneous bandwidth changes. `SCHED_DEADLINE` does not wait for the zero-lag time before allowing a task to modify its parameters. `SCHED_DEADLINE` will immediately enact any change in parameters requested with `sched_setattr()` so long as the task's resulting `dl_bw` does not violate (4.2). While the task's static parameters (*e.g.*, `dl_runtime`, `dl_period`, `dl_bw`) are changed immediately, the task's current `runtime` and `deadline` are unchanged. This can be exploited as follows to result in unbounded response times.

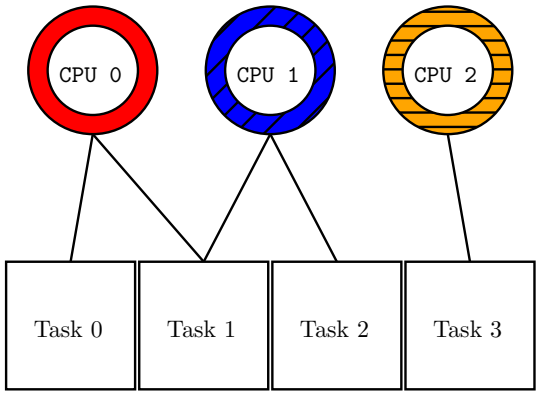
▼ **Example 4.9.** Consider the schedule in Figure 4.11. We assume the tasks in this example never suspend. The system contains one CPU and begins with no tasks. At time 0, Task 0 requests to enter `SCHED_DEADLINE` with $(\texttt{dl\_runtime}, \texttt{dl\_period}) = (94, 100)$. This task is accepted because doing so will not violate (4.2). Task 0 begins executing at time 0 with $\texttt{runtime} = 94$ and $\texttt{deadline} = 100$, before immediately requesting `dl_runtime` to be changed to 1. Task 0's `runtime` and `deadline` remain as 94 and 100 after this change. At time 10, Task 1 requests to enter `SCHED_DEADLINE` with $(\texttt{dl\_runtime}, \texttt{dl\_period}) = (94, 100)$. This request is accepted by the ACS because Task 0 reduced its bandwidth. Task 1 begins executing with $\texttt{runtime} = 94$ and $\texttt{deadline} = 110$. After this point, both tasks alternate executing on the CPU.

However, prior to when Task 0 returns from the throttled state, Task 1 changes its `dl_runtime` to 1 and Task 0 changes its `dl_runtime` to 94. When Task 0 becomes ready at time 100, its `runtime` and `deadline` are determined entirely by Task 0's `dl_runtime` and `dl_period` at the instant it becomes ready. Thus, the `runtime` of Task 0 is set to 94. Likewise, prior to when Task 1 is replenished at time 188 (recall that a task that exhausts `runtime` after its `deadline` is immediately replenished), Task 0 sets its `dl_runtime` to 1 such that Task 1 can set its `dl_runtime` to 94. As the total bandwidth of the system technically never exceeds 0.95, all requests are accepted by the ACS.

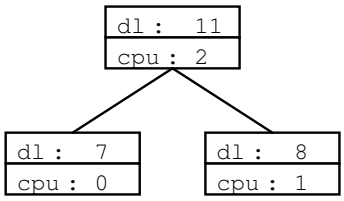
If Tasks 0 and 1 continue taking turns with having $\texttt{dl\_runtime} = 94$, then every `runtime` replenishment in this system occurs as if both tasks always have `dl_bw` of 0.94. Because the CPU only has a capacity of 1.0, the system is overloaded and results in unbounded response times even though (4.2) is never violated. ▲

That the behavior in Example 4.9 is possible is documented in the `SCHED_DEADLINE` source code. Bounded response times can be restored by exhausting the `runtime` of any task that changes its parameters via `sched_setattr()` and delaying the change in `total_bw` in the corresponding `root_domain`'s `dl_bw` until the zero-lag time.

`sched_dl_global_validate()` bug. `sched_dl_global_validate()` is unique in that it does not call `__dl_overflow()` to check if (4.2) is maintained. `sched_dl_global_validate()` does not appear to consider that CPUs may have asymmetric capacities. This seems to be a bug that permits `sched_rt_runtime_us` to be set to lower values than would otherwise be permitted by the ACS. This



(a) Affinity graph.



(b) `cpudl` heap at time 6.

Figure 4.12: Example 4.10 illustrations.

seems like an oversight that can be easily remedied by modifying `sched_dl_global_validate()` to call `__dl_overflow()`.

4.4.5 Affinities

As stated in Section 4.4.4, disabling the ACS allows arbitrary affinities to be set for `SCHED_DEADLINE` tasks. `SCHED_DEADLINE` does not follow Weak-APA-EDF when the ACS is disabled. This is because the `cpudl`, which is used by `SCHED_DEADLINE` to optimize computing target CPUs when pushing tasks, is oblivious to affinities, so tasks may be left unscheduled even when they have affinity for CPUs executing tasks with later deadlines. This is demonstrated in the following example.

▼ **Example 4.10.** Consider three implicit-deadline tasks executing on three CPUs such that tasks' affinities are as in Figure 4.12a. All three CPUs belong to a single `root_domain`. Let the `dl_runtime` and `dl_period` of Tasks 0, 2, and 3 be 3 and 6, respectively. For Task 1, its `dl_runtime` and `dl_period` are both 2.

A schedule of this system is illustrated in Figure 4.13. Task 1 never suspends and initially executes on CPU 0. Even though Task 0, which only has affinity for CPU 0, enters the system at time 1, Task 1 does not migrate until Task 0 has an earlier deadline at time 6. However, prior to Task 1's attempt to migrate, Tasks 2 and 3 enter the system such that all CPUs execute tasks at time 6. The `cpudl` heap of the `root_domain` at time 6 is illustrated in Figure 4.12b. CPU 2 with deadline 11 is at the root of the heap. Thus, `SCHED_DEADLINE` will attempt to push Task 1 onto CPU 2. Because Task 1 does not

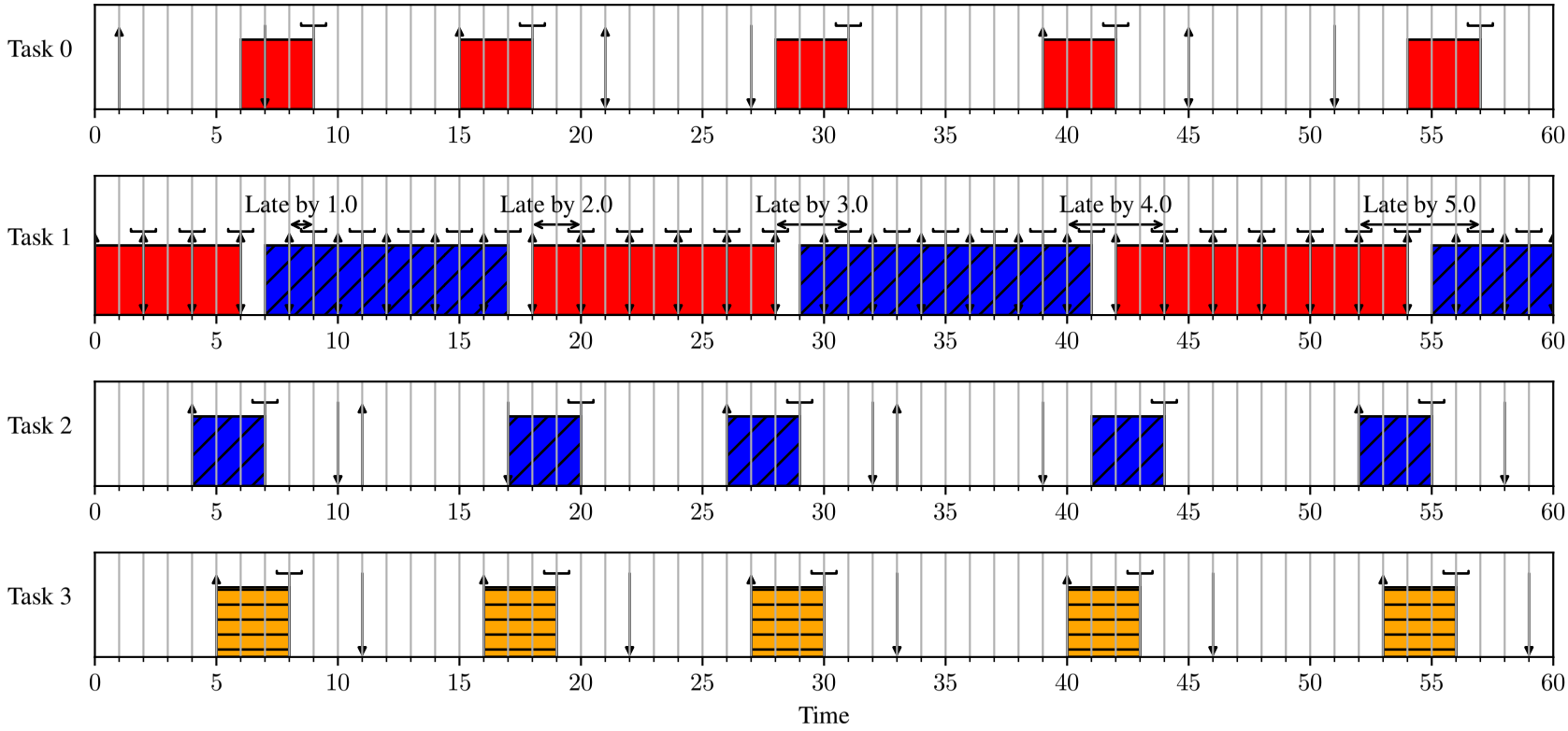


Figure 4.13: Example 4.10 schedule.

have affinity for CPU 2, this push will fail, and Task 1 is not scheduled. This is despite Task 1 having an earlier deadline than Task 2, which is executing on CPU 1, thereby violating Weak-APA-EDF. Task 1 remains unscheduled until it is pulled by CPU 1 at time 7.

Weak-APA-EDF is similarly violated at time 17, except at this time Task 2, rather than Task 0, forces Task 1 to attempt to migrate. As illustrated in Figure 4.13, this pattern can be repeated infinitely often, and with each occurrence, the maximum response time experienced by Task 1 increases by 1.0.▲

Note that Example 4.10 is unrealistic in that unbounded response times for the considered task system would likely not occur in a real system. The example relies on contrived waking times for Tasks 0, 2, and 3 in order to force Task 1 to reference the `cpudl` heap. Even though the presented example is unlikely to be observed in practice, its existence is problematic because it proves that response-time bounds cannot be guaranteed.

4.4.6 Asymmetric Capacities

Function (or macro on some architectures) `arch_scale_cpu_capacity()` takes a CPU index as its argument and returns said CPU's capacity. When decrementing a task's `runtime` value in `update_curr_dl()`, if flag `SCHED_FLAG_RECLAIM` is unset, execution duration `delta_exec` is multiplied by the capacity of the CPU the task executed on.

Origin. The capacities of the CPUs on a given platform are derived from empirical per-CPU values in the devicetree (*i.e.*, `.dtb` and `.dts` files), a standard file for providing hardware information to the OS. In the devicetree, each CPU is given a `capacity-dmips-mhz` value that is proportional to its Linux capacity. DMIPS/MHz is an instructions-per-second performance measurement of the CPU on the Dhrystone (Weicker, 1984) benchmark when the CPU operates at a fixed 1 MHz frequency. The Linux capacity for a CPU is computed from the product of maximum frequency (in MHz) and `capacity-dmips-mhz`. These products are then normalized such that the greatest capacity of any CPU is 1024. 1024 should be thought of as the value 1.0 (*i.e.*, full capacity), only left-shifted by `SCHED_CAPACITY_SHIFT` (*i.e.*, 10) such that fractional capacities can be represented as integers.

Migration logic. `SCHED_DEADLINE` does not implement Ufm-EDF in the presence of asymmetric capacities. The extent of `SCHED_DEADLINE`'s migration logic with respect to asymmetric capacities is that `SCHED_DEADLINE` will attempt to avoid migrating tasks to CPUs with low capacity (relative to the tasks

being migrated). Specifically, SCHED_DEADLINE considers asymmetric capacities in its migration logic in function find_later_rq(), which is used by push_dl_task() and select_task_rq() to find target runqueues. Recall that the behavior of this function was separated into cases in Section 4.4.2. Asymmetric capacities are considered when the intersection between cpus_mask of the task of interest (either a pushed task or a waking task) and free_cpus in the corresponding root_domain is non-empty, *i.e.*, there are CPUs in the root_domain's span that the task of interest has affinity for and have no SCHED_DEADLINE tasks. When this intersection is non-empty, find_later_rq() attempts to return a CPU with capacity at least the task's density dl_density. If no CPUs in the intersection have capacity greater than dl_density, the CPU with the greatest capacity in this intersection is returned.

Response time bounds. The ACS does not guarantee bounded response times under asymmetric capacities. Consider a root_domain containing two tasks with dl_bw of 0.95 (ignoring the shift by BW_SHIFT) and one full-capacity CPU with arbitrarily many CPUs with capacity 0.5 (ignoring the shift by SCHED_CAPACITY_SHIFT). Because tasks cannot execute in parallel with themselves, these two tasks are executed on at most two CPUs at any time. Even though this root_domain is accepted by the ACS (total_bw is 1.9 while total capacity is arbitrarily large), these two tasks can only consume 1.5 units of capacity. As total_bw exceeds the capacity provided to the tasks, the tasks would have unbounded response times (assuming they do not suspend).

4.4.7 Priority Inheritance

This and the remaining subsections in this section pertain to features of SCHED_DEADLINE or Linux that are incompatible with the analysis in this dissertation. As such, these features are discussed at a higher level than the previous subsections. We start with priority inheritance.

When a task inherits a SCHED_DEADLINE task's priority, the inheriting task's pi_se member is set to point to the inherited from task's sched_dl_entity by rt_mutex_setprio() (recall Listing 4.10). This has two effects on the inheriting task: said task behaves as if its static parameters (*e.g.*, dl_deadline, dl_runtime, dl_period) are replaced with the inherited-from task's parameters and said task always bypasses throttling.

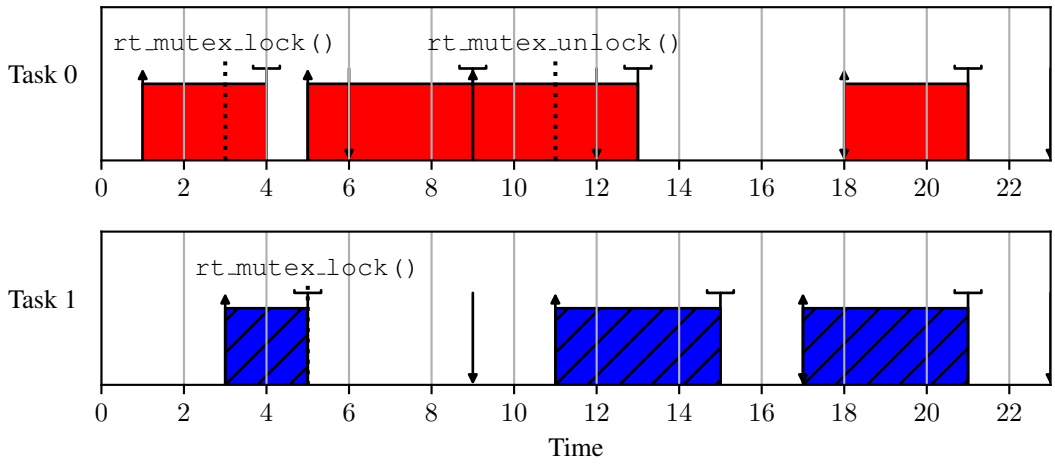


Figure 4.14: Priority inheritance.

As an example of static parameters being replaced, recall Listing 4.34 of `replenish_dl_entity()`. Observe that `deadline` and `runtime` are incremented according to the `dl_period` and `dl_runtime` of the `pi_se` task.

Throttling is bypassed for priority-inheriting tasks because they hold mutexes being waited on by higher-priority tasks. Tasks that would otherwise be throttled may reach the end of their critical sections earlier and release their mutexes.

As a result of bypassing throttling, a priority-inheriting task is always immediately re-enqueued with `ENQUEUE_REPLENISH` in `update_curr_dl()` whenever it is dequeued due to depleting `runtime`. This has an effect analogous to early releasing (see Definition 2.11 in Section 2.1) in that the task consumes the budget of what would otherwise be future replenishments. As in early releasing, deadlines are set based on the original future replenishment times (*i.e.*, original arrival times), and not based on when budget is consumed (*i.e.*, early release times).

▼ **Example 4.11.** Consider two implicit-deadline tasks executing on two CPUs as illustrated in Figure 4.14. Task 0 has `dl_runtime` of 3 and `dl_period` of 5, while Task 1 has `dl_runtime` of 4 and `dl_period` of 6.

Tasks 0 and 1 share some RT-mutex. Task 0 locks this mutex at time 3. Because no other task is waiting on the mutex, Task 0's `pi_se` pointer remains pointed at Task 0.

At time 5, Task 1 attempts to acquire the mutex. This sets the `pi_se` pointer of Task 0, the holder of the mutex, to Task 1. `rt_mutex_setprio()` enqueues (line 24 of Listing 4.10) the previously

throttled Task 0. Task 0 receives 4 units of `runtime`, which is Task 1's `dl_runtime` value. Task 0's deadline is incremented from 6 to 12, in accordance with Task 1's `dl_period` value.

Task 0 continues early releasing in this fashion until it releases the mutex at time 11. This resets its `pi_se` pointer to itself. When Task 0 exhausts its `runtime` at time 13, it is throttled until time 18, its next replenishment time.

At time 18, Task 0 receives 3 units of `runtime` and its `deadline` is incremented by 5 time units, as per its original `dl_runtime` and `dl_period`. ▲

4.4.8 GRUB

GRUB (short for Greedy Reclaimation of Unused Bandwidth) reduces the budget consumption rate of tasks. Analytically, this is similar to reducing execution speed. This reduction allows each CBS to run for longer, which is desirable for tasks that occasionally overrun their budgets. The reduction in consumption must be limited to prevent `SCHED_DEADLINE` tasks from consuming more than `sched_rt_runtime_us`/`sched_rt_period_us` of capacity, as guaranteed by the ACS.

Unused bandwidth. The idea behind GRUB is that the farther the total bandwidth of active tasks is from `sched_rt_runtime_us`/`sched_rt_period_us` of capacity, the more the budget consumption rate can be reduced. The difference between the total active bandwidth and this fraction of capacity is the *unused bandwidth*.

There exist several GRUB variants that differ by how unused bandwidth is reclaimed. The GRUB variant used in Linux is a combination of the sequential and parallel reclaiming variants proposed by Abeni et al. (2016). Unused bandwidth is subdivided between the `dl_rq`s, with the amount of unused bandwidth allocated to a given `dl_rq` determining the rate of budget consumption for tasks on that `dl_rq`. Unused bandwidth is accounted for in members `this_bw` and `extra_bw`. `this_bw` is like `running_bw` except the bandwidths of inactive tasks that suspended while on this `dl_rq` are also counted (note that because these tasks are inactive, they are not *on* the `dl_rq`). The unused bandwidth from inactive tasks is the *inactive* bandwidth, and is computed on a `dl_rq` as `this_bw` − `running_bw`.

The remaining unused bandwidth is due to the difference between the total bandwidth (both active and inactive) and `sched_rt_runtime_us`/`sched_rt_period_us` of capacity in the `root_domain` corresponding with the runqueues of interest. This is called the *extra* bandwidth. Extra bandwidth is computed

from the root_domain's dl_bw member. For a dl_rq dl with root_domain rd that has a span of cpus CPUs, dl->extra_bw is equivalent to sched_rt_runtime_us/sched_rt_period_us − rd.dl_bw.total_bw/cpus.

Note that extra_bw is computed using the number of CPUs. This assumes that all CPUs have equal capacity. GRUB should not be enabled on systems with asymmetric capacities.

Decreasing runtime under GRUB. GRUB relies on two additional dl_rq members, max_bw and bw_ratio. max_bw is set to the fraction sched_rt_runtime_us/sched_rt_period_us and bw_ratio is set to the inverse of this fraction.

GRUB is used when flag SCHED_FLAG_RECLAIM is set. Under GRUB, in update_curr_dl(), the execution duration delta_exec is multiplied by some factor to yield scaled_delta_exec. This factor is bw_ratio multiplied by

$$\max\left\{\texttt{dl\_bw}, \texttt{max\_bw} - (\texttt{this\_bw} - \texttt{running\_bw}) - \texttt{extra\_bw}\right\}.$$

Maintaining this_bw. For a given dl_rq, this_bw is set by functions __add_rq_bw() and __sub_rq_bw(), while running_bw is set by __add_running_bw() and __sub_running_bw(). __add_rq_bw() and __add_running_bw() are presented below.

```
void __add_rq_bw(u64 dl_bw, struct dl_rq *dl_rq)
{
    dl_rq->this_bw += dl_bw;
}

void __add_running_bw(u64 dl_bw, struct dl_rq *dl_rq)
{
    dl_rq->running_bw += dl_bw;
    cpufreq_update_util(rq_of_dl_rq(dl_rq), 0);
}
```

__sub_rq_bw() and __sub_running_bw() are identical outside of replacing addition with subtraction. These functions also include checks for overflow and improper locking that are omitted from the above listing. Function cpufreq_update_util() alerts the frequency scaling governor that the utilization

on a runqueue has changed, and will be discussed in Section 4.4.9. Function `rq_of_dl_rq()` uses `container_of()` to return a pointer to a `dl_rq`'s containing `rq`.

Recall that `this_bw` is the total bandwidth of all tasks that last migrated to the corresponding runqueue (such tasks may not be queued on this runqueue). `__add_rq_bw()` is primarily called from within `enqueue_task_dl()` when the task is enqueued as part of a migration (*e.g.*, a push, pull, or waking on a different CPU). That `enqueue_task_dl()` was called as part of a migration can be determined by observing if `on_rq` is `TASK_ON_RQ_MIGRATING` or if the flag `ENQUEUE_MIGRATED` is set.

`__sub_rq_bw()` is similarly primarily called by `dequeue_task_dl()` when `on_rq` is `TASK_ON_RQ_MIGRATING`. Note that in the case of a suspension, `dequeue_task_dl()` will not call `__sub_rq_bw()` because `on_rq` will have value 0. This is valid because `this_bw` includes suspended tasks until such tasks are migrated away. If a suspended task is migrated during wakeup, function `migrate_task_rq_dl()` calls `__sub_rq_bw()` on the previous runqueue. Recall that `migrate_task_rq()` is called whenever a task moves to a different runqueue. `migrate_task_rq()` is called before `__set_task_cpu()` and `enqueue_task()`, thus `task_rq()` returns the previous runqueue when called from `migrate_task_rq_dl()`.

Maintaining `running_bw`. `running_bw` corresponds to the subset of tasks accounted for in `this_bw` that are active. Thus, while a task is active and migrated, `__add_running_bw()` and `__sub_running_bw()` are also called when `__add_rq_bw()` and `__sub_rq_bw()` are called, respectively.

`__sub_running_bw()` and `__add_running_bw()` must also be called when a task becomes inactive and active, respectively. A task becomes inactive by suspending and passing its zero-lag time. For a CBS represented by a `sched_dl_entity`, the zero-lag time is computed as

$$\texttt{deadline} - \texttt{runtime} \cdot \frac{\texttt{dl\_period}}{\texttt{dl\_runtime}}. \tag{4.4}$$

That `dequeue_task_dl()` is called due to a suspension is determined by observing if flag `DEQUEUE_SLEEP` is set. If `DEQUEUE_SLEEP` is set, then `dequeue_task_dl()` calls `__sub_running_bw()` if the zero-lag time has passed. If the zero-lag time is in the future, `dequeue_task_dl()` arms `inactive_timer` to fire at the zero-lag time. The callback function of `inactive_timer`, `inactive_task_timer()`, calls `__sub_running_bw()` when triggered. An inactive task becomes active by waking, in which case `enqueue_task_dl()` will call `__add_running_bw()`.

Note that the zero-lag time formula in (4.4) is not equivalent to Definition 2.12 for non-implicit-deadline tasks. `deadline` should be replaced with the next replenishment time `deadline` − `dl_deadline` + `dl_period` to make (4.4) equivalent to Definition 2.12. Because these zero-lag time formulas are distinct, GRUB may cause additional deadline misses on single-CPU systems.

`enqueue_task_dl()` must be able to determine whether a waking task was previously inactive or active and waiting for its zero-lag time. For the latter, `enqueue_task_dl()` must not call `__add_running_bw()`, as `__sub_running_bw()` has not been called on this task (due to delaying the calling of `__sub_running_bw()` until the zero-lag time). `enqueue_task_dl()` makes this distinction by observing the `dl_non_contending` flag, which is set by `dequeue_task_dl()` when waiting for the zero-lag time and cleared by `enqueue_task_dl()` and `inactive_task_timer()`.

4.4.9 DVFS

DVFS with `SCHED_DEADLINE` tasks is based on the GRUB-PA algorithm (Scordino et al., 2018). While the GRUB variant discussed in Section 4.4.8 reduces budget consumption to improve quality of service to tasks, GRUB-PA reduces budget consumption to reflect that tasks executed at lower frequencies. The main idea of GRUB-PA is to set the frequency of a CPU to the maximum frequency multiplied by the total active bandwidth of its runqueue (*i.e.*, `running_bw` on the CPU's corresponding `dl_rq`). If the total active bandwidth exceeds 1.0, the frequency is capped at the maximum frequency.

Function (or macro on some architectures) `arch_scale_freq_capacity()` takes a CPU index and returns the ratio of the current frequency divided by the maximum frequency of said CPU. Note this ratio is also left-shifted by `SCHED_CAPACITY_SHIFT` to be represented as an integer. When decrementing `runtime` for a task in `update_curr_dl()`, if flag `SCHED_FLAG_RECLAIM` is not set, the execution duration `delta_exec` is multiplied by the frequency ratio prior to multiplying by the CPU's capacity.

GRUB-PA and unbounded response times. As originally proposed, GRUB-PA Scordino et al. (2018) can lead to unbounded response times, as will be demonstrated in the following example. Note that we discuss the original GRUB-PA algorithm and not its more complicated implementation in the `schedutil` governor. It is difficult to model `schedutil` for an example due to its complexities, which will be discussed in later paragraphs.

▼ **Example 4.12.** Consider three implicit-deadline tasks running on two CPUs such that each task has `dl_runtime` of 3 and `dl_period` of 5. Both CPUs have capacity of 1.0 and equivalent maximum frequency. No task suspends after entering `SCHED_DEADLINE`. Task 0 enters the system at time 0, Task 1 at time 1, and Task 2 at time 2.

A schedule of this system is illustrated in Figure 4.15. Initially, when Task 0 enters the system at time 0, it is the only task contributing to CPU 0's `running_bw`, which is 3/5. Thus, CPU 0's frequency is scaled to 3/5 of the maximum frequency. CPU 0 continues executing at $3/5 = 0.6$ of its maximum frequency starting form time 0. Over time interval $[0, 2)$, Task 0 is executed for 2 time units on CPU 0, which delivers 0.6 units of capacity per time unit. Task 0 has $\texttt{dl\_runtime} - 0.6 \cdot 2 = 3 - 1.2 = 1.8$ units of `runtime` remaining.

At time 1, Task 1 enters the system. It either enters already on CPU 1 or is pushed from CPU 0 to CPU 1 when it enters the system. As with Task 0 on CPU 0, Task 1 is the only task contributing to CPU 1's `running_bw`, and so CPU 1's frequency is also scaled to $3/5 = 0.6$ of its maximum. Over time interval $[1, 6)$, Task 1 consumes $0.6 \cdot (6 - 1) = 3 = \texttt{dl\_runtime}$ units of `runtime`. Task 1 exhausts its `runtime` exactly at its `deadline`, 6. Task 1 continues to exhaust its `runtime` exactly at its `deadline` every 5 time units until it is preempted at time 36.

At time 2, Task 2 enters the system. Assume it enters on CPU 0's runqueue. Because two tasks now contribute to CPU 0's `running_bw`, which becomes $6/5 > 1.0$, CPU 0's frequency is scaled to its maximum. Task 0, with 1.8 units of `runtime` remaining at time 2, exhausts its `runtime` at time $2 + 1.8/1.0 = 3.8$ (CPU 0 provides its maximum capacity, 1.0, at its maximum frequency). Task 0 and Task 1 continue alternating executing on CPU 0 until CPU 1 pulls Task 0 at time 36.

Once Task 0 is pulled to CPU 1 at time 36, the `running_bw` of both CPUs changes. CPU 1 scales its frequency to 1.0 of its maximum and CPU 0 scales its frequency to 0.6 of its maximum. Because there are three tasks and only two CPUs, there is always a CPU executing at 0.6 of its maximum frequency. The task executing on this CPU can never reduce the response times of successive jobs because the capacity delivered by this CPU is equal to the task's bandwidth. This can be observed for Task 1 over $[1, 36)$, during which Task 1 always exhausts `runtime` at its `deadline`.

Whatever CPU is not executing at 0.6 of its maximum frequency executes at its maximum frequency. This is not sufficient for the two tasks that contribute to said CPU's `running_bw`, which equals

$2 \cdot 3/5 = 1.2$. Because this CPU delivers at most 1.0 units of capacity per time unit, successive jobs of these two tasks will increase in response time. This can be observed for Tasks 0 and 2 over $[0, 36)$. ▲

Frequency invariance and `clock_pelt`. The implementation of GRUB-PA in the `schedutil` governor differs due to the presence of non-`SCHED_DEADLINE` tasks in the system. Failing to account for these tasks in GRUB-PA could result in these tasks being starved as higher-priority `SCHED_DEADLINE` tasks executing at reduced frequencies consume the entire system's capacity. `schedutil` requires bandwidth information to account for these non-`SCHED_DEADLINE` tasks. Because Linux is not given the bandwidths of non-`SCHED_DEADLINE` tasks, it must estimate them by measuring the fraction of time such tasks occupy the CPUs.

For each CPU, Linux maintains an estimate of the total bandwidth of tasks belonging to `fair_sched_class` and `rt_sched_class`. These bandwidth estimates are added to `running_bw` when performing GRUB-PA on each CPU. Estimating bandwidths in the presence of frequency scaling results in a circular dependency: bandwidth estimates are used to scale CPU frequencies and scaling frequencies affects measured bandwidths. This circular dependency is resolved by normalizing bandwidth estimates, *i.e.*, scaling bandwidth using the corresponding CPU's frequency to compute the expected bandwidth at maximum frequency.

Applying this normalization to the bandwidth requires knowledge of the current CPU frequency. The current frequency can be read from the hardware (*i.e.*, by reading specific registers) or from the CPU frequency driver (`arch_scale_freq_capacity()` observes these sources to return a frequency ratio).[14] Normalizing bandwidth by reading from the hardware is called *frequency-invariant* bandwidth estimation. Frequency-invariant bandwidth estimation makes use of `clock_pelt`.

Recall the discussion in Section 4.4.3 comparing `clock` and `clock_task`. `clock_pelt` is another measure of the current time derived from `clock_task` (recall that `clock_task` is itself derived from `clock`). When frequency is scaled down, the timeline observed by `clock_pelt` slows the rate that time advances. The timeline catches up to `clock_task` when the CPU idles. A simplified example of the use of `clock_pelt` in frequency-invariant bandwidth estimation is provided below.

▼ **Example 4.13.** Consider two `rt_sched_class` tasks executing on a single CPU. At maximum frequency, Task 0 executes for 1 time unit every 5 time units and Task 1 executes for 1 time unit every 7

[14] Note that reading the current frequency from the frequency driver is generally considered unreliable compared to reading from the hardware. Reading from the hardware requires architectural support.

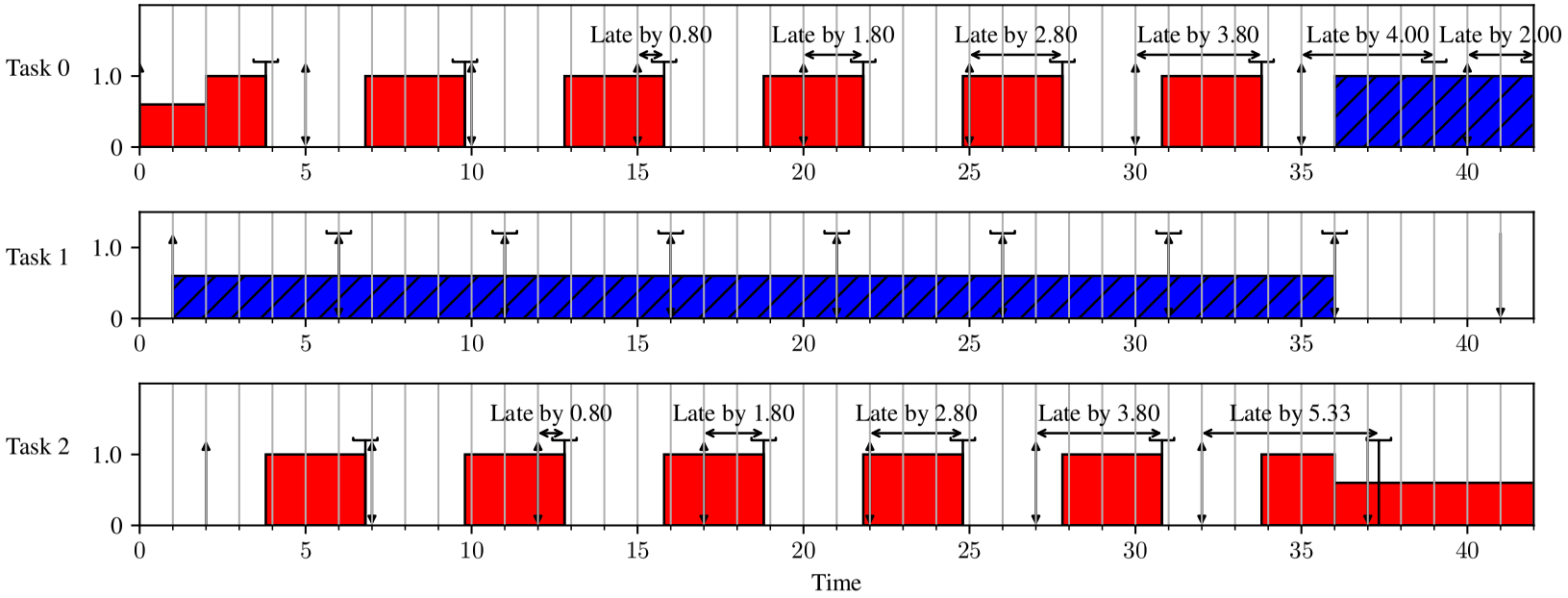


Figure 4.15: GRUB-PA schedule.

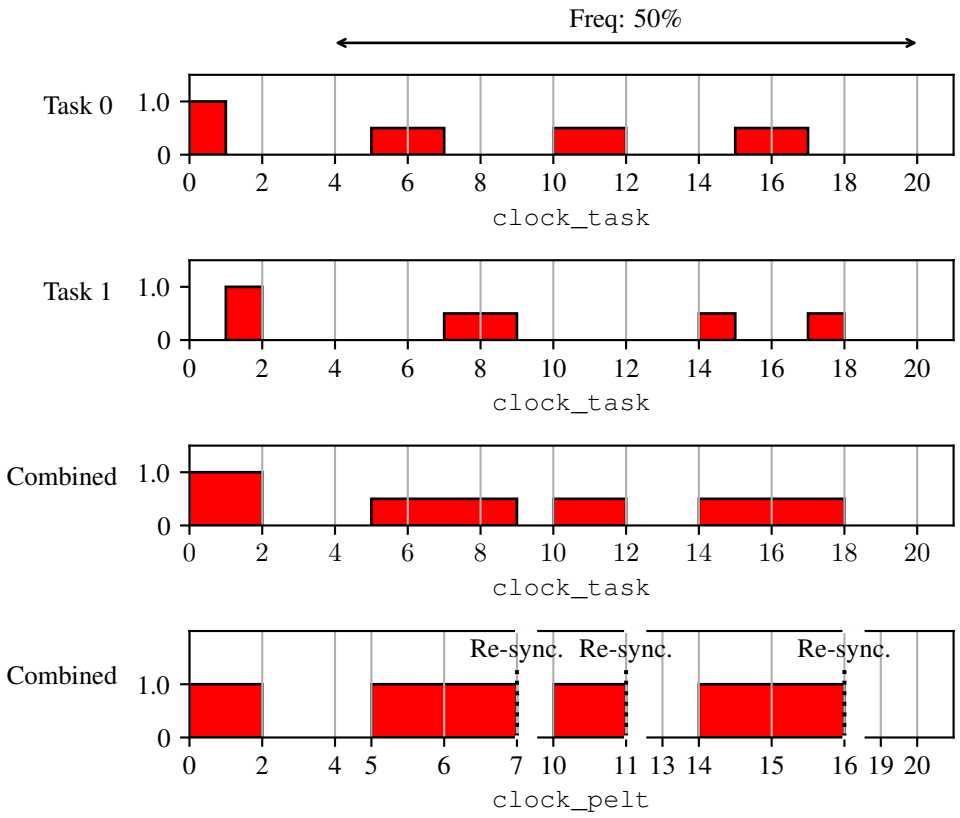


Figure 4.16: `clock_pelt` example.

time units. A schedule of these two tasks is illustrated in Figure 4.16. The tasks are illustrated individually in the first two rows, and then their combined execution is illustrated in the last two rows. The first three rows share a common time axis as observed by `clock_task`, while the last row has a time axis as observed by `clock_pelt`.

Suppose that at time 4, the CPU scales its frequency to 50% of its maximum. This causes the timelines observed by `clock_task` and `clock_pelt` to diverge. The interval represented by $[5, 9)$ by `clock_task` is represented by $[5, 7)$ by `clock_pelt` because `clock_pelt` advances time at 50% of the rate of `clock_task` while the frequency is scaled to 50%. `clock_pelt` and `clock_task` re-synchronize at the end of this interval when the CPU idles. The two clock values similarly diverge at times 10 and 14.

Consider how the total bandwidth of these tasks is computed when `clock_task` and `clock_pelt` are used. When observing `clock_task`, the two tasks executed over $[0, 2)$, $[5, 9)$, $[10, 12)$, and $[14, 18)$. The two tasks executed for a total of $(2 - 0) + (9 - 5) + (12 - 10) + (18 - 14) = 2 + 4 + 2 + 4 = 12$ time units over the 20 time units illustrated in the schedule, *i.e.*, a measured total bandwidth of $\frac{\text{total execution time}}{\text{observed time}} = \frac{12}{20} = 60\%$. Compare this against the actual total bandwidth of Tasks 0 and 1, $\frac{1}{5} + \frac{1}{7} \approx 34\%$.

When observing `clock_pelt`, the two tasks executed over $[0, 2)$, $[5, 7)$, $[10, 11)$, and $[14, 16)$. The two tasks executed for a total of $(2-0)+(7-5)+(11-10)+(16-14) = 2+2+1+2 = 7$ time units over the 20 illustrated time units, *i.e.*, a measured bandwidth of $\frac{7}{20} = 35\%$. This is much closer to the true total bandwidth of Tasks 0 and 1 (34%) than the total bandwidth measured from `clock_task`. ▲

Under frequency-invariant bandwidth estimation, bandwidth normalization is inherent due to the usage of `clock_pelt` during measurement. If the frequency cannot be read from the hardware, the bandwidth is normalized by multiplying the measured bandwidth by the ratio of the current and max frequencies. This reduces the bandwidth estimation when frequency is scaled down.

GRUB-PA in `schedutil`. `schedutil` computes a total active runqueue bandwidth for each CPU by adding the bandwidth estimates for `fair_sched_class` and `rt_sched_class` tasks to `running_bw`. How `schedutil` sets CPU frequency depends on if frequency-invariant bandwidth estimation is

supported. If supported, the frequency of a CPU is set to

$$1.25 \cdot \text{max frequency} \cdot \frac{\text{active runqueue bandwidth}}{\text{CPU's capacity}}. \tag{4.5}$$

This product is capped at the maximum frequency. The frequency computed by (4.5) differs from the original GRUB-PA algorithm (Scordino et al., 2018) because of the leading 1.25 constant and the division by the CPU's capacity. The constant 1.25 seems to be a magic number without a basis in theory. The original GRUB-PA algorithm does not consider asymmetric capacities. The active runqueue bandwidth is divided by the CPU's capacity to increase frequencies set for low-capacity CPUs. For example, a CPU with `running_bw` of 0.5 and a capacity of 0.5 should run at maximum frequency.

If frequency-invariant bandwidth estimation is not supported, the frequency is instead set to

$$1.25 \cdot \text{current frequency} \cdot \frac{\text{non-invariant active runqueue bandwidth}}{\text{CPU's capacity}}. \tag{4.6}$$

The rational behind using the current frequency is that the frequency-invariant active runqueue bandwidth is approximated by multiplying the measured bandwidth by the ratio of the current and maximum frequencies. Combining (4.5) with this approximation yields (4.6).

Note that (4.6) breaks GRUB-PA for `SCHED_DEADLINE` tasks. The contribution of `running_bw` to the active runqueue bandwidth is the total bandwidth of `SCHED_DEADLINE` tasks. Recall that the bandwidths of `SCHED_DEADLINE` tasks are derived from parameters `dl_runtime` and `dl_period`, which are provided to Linux during a policy change request. Because `dl_runtime` and `dl_period` are not determined by measurement, they are invariant to CPU frequencies. This contribution by `running_bw` should be multiplied by the maximum frequency regardless of whether frequency invariance is supported. Multiplying by the current frequency in (4.6) incorrectly reduces frequency. In practice, this probably does not cause additional deadline misses due to the 1.25 scaling factor.

Triggering frequency updates. Frequency updates under `schedutil` are triggered by calling `cpufreq_update_util()`. Recall from Section 4.4.8 that this function is called by `__add_running_bw()` and `__sub_running_bw()`, the helper functions for modifying `running_bw` on the corresponding CPU. Thus, `schedutil` attempts to set the CPU frequency whenever `running_bw` is modified on a CPU.

`cpufreq_update_util()` calls a callback function set by the selected frequency governor. For `schedutil`, the callback function is one of `sugov_update_single_perf()`, `sugov_update_single_freq()`, or `sugov_update_shared()`. These callback functions communicate with the frequency driver. `sugov_update_single_perf()` and `sugov_update_single_freq()` differ in how they communicate with the driver. These differences are beyond the scope of this dissertation.

`sugov_update_shared()` is used when the corresponding CPU is in a `cpufreq_policy` that contains other CPUs. A `cpufreq_policy` is a set of CPUs that share hardware for controlling frequency. `sugov_update_shared()` scales the frequencies of all CPUs in the `cpufreq_policy` according to the largest total bandwidth of any CPU in the `cpufreq_policy`.

4.4.10 Core Scheduling

Core scheduling is optionally enabled under SMT. In Linux terms, a *core* is a set of neighboring logical CPUs under SMT. Under core scheduling, tasks are assigned tags called *cookies* such that the scheduler will only schedule tasks with the same cookie on the CPUs in a core. A task's cookie is stored in member `core_cookie` in `struct task_struct`. A new cookie can be generated for a task and a cookie can be shared between tasks using the `prctl` (process control) system call. Note that a forked task automatically inherits its parent's cookie. By default, each task's cookie has value 0. Kernels configured with core scheduling add additional members (presented in Listing 4.36) to `task_struct` and `rq`. The usage of these member is detailed in the following paragraphs.

For each core, one `rq` corresponding with a CPU in said core is designated the *core runqueue* that stores the state of said core. This state is the "shared state" members in Listing 4.36. The most important of these members if `core_cookie`, the cookie value of the core. `core_cookie` in the core runqueue matches `core_cookie` in `task_struct` for any task scheduled on a CPU in the core. For each runqueue of a CPU in a core, pointer `core` points to the core runqueue storing the shared state. The remaining shared state members are `core_task_seq` and `core_pick_seq`. These are sequence numbers that are used with per-runqueue members `core_pick` and `core_sched_seq` to implement coordinated scheduling with the core, which will be detailed in later paragraphs. The final members we discuss are `core_tree` in `rq` and `core_node` in `task_struct`. `core_node` is used to insert a task into `core_tree`. The usage of `core_tree` will be described in later paragraphs.

```
struct rq {
    ⋮
    /* per rq */
    struct rq           *core;
    struct task_struct  *core_pick;
    unsigned int    core_sched_seq;
    struct rb_root      core_tree;

    /* shared state */
    unsigned int    core_task_seq;
    unsigned int    core_pick_seq;
    unsigned long   core_cookie;
};

struct task_struct {
    ⋮
    struct rb_node      core_node;
    unsigned long       core_cookie;
};
```

Listing 4.36: Core scheduling members.

Coordinated scheduling. Under core scheduling, the `__schedule()` function behaves differently than the pseudocode presented in Listing 4.4. Specifically, the logic represented by lines 16-28 of Listing 4.4 are replaced with the coordinated scheduling logic to be described in this subsection.

CPUs in a core must coordinate their scheduling to only schedule tasks with the same cookie. To facilitate this, any CPU that calls `__schedule()` suggests to the other CPUs in the core which tasks they should schedule. The CPU calling `__schedule()` writes these suggested tasks to the `core_pick` pointers in the other CPUs' runqueues.

Let a CPU calling `__schedule()` be denoted as a rescheduling CPU. Core scheduling in `__schedule()` occurs in three steps.

Core Sched. 1: Decide whether to schedule a suggested task.

In the first step, the rescheduling CPU observes its corresponding `core_pick` pointer to see if another CPU in the core has already suggested a task from within its own previous call to `__schedule()`. If `core_pick` is valid (how `core_pick` may become invalid will be explained later when discussing sequence numbers), then `core_pick` is scheduled and the second and third steps are skipped.

Core Sched. 2: Compute cookie for the core.

The second step involves selecting `core->core_cookie` for the core. The rescheduling CPU identifies the highest priority task on any of the runqueues corresponding with the core. This is done by iterating over the CPUs in the core and calling `pick_task()` (for each `sched_class`) to return the highest priority task on said CPUs runqueue. `core->core_cookie` is set to the highest-priority task's `core_cookie`.

Core Sched. 3: Pick tasks for CPUs in the core matching the cookie.

The third step computes `core_pick` for each CPU in the core such that `core_pick->core_cookie` matches `core->core_cookie`. Again, the CPUs in the core are iterated over, though this time only tasks with `core_cookie` matching `core->core_cookie` are considered. For the CPU in a given iteration, `core_pick` is set to the highest-priority task with a matching cookie.

The rescheduling CPU, which is currently calling `__schedule()`, straightforwardly schedules its corresponding `core_pick` task. The rescheduling CPU alerts the other CPUs to reschedule by calling `resched_curr()`. This causes the other CPUs to call `__schedule()`, during which they will schedule their corresponding `core_pick` tasks in Core Sched. 1.

Cookie search tree. Core Sched. 3 requires efficiently searching for high-priority tasks matching a given cookie. For this purpose, each `task_struct` that has non-zero `core_cookie` and is enqueued onto a runqueue is also added to that runqueue's `core_tree`, a binary search tree ordered on `core_cookie`. Ties between tasks with equal `core_cookie` are broken such that higher-priority tasks are to the left of lower-priority tasks. This simplifies searching for the highest-priority task with a certain cookie.

▼ **Example 4.14.** Consider a runqueue containing Tasks 0-6 with values as illustrated in Figure 4.17. Figure 4.17 illustrates the structure of `core_tree` for this runqueue. `core_tree` is ordered first by `core_cookie` (*e.g.*, Task 6 with `core_cookie` of 5 is to the right of Tasks 0 and 1 with `core_cookie` of 3) and second by task priority (*e.g.*, Task 0 with `deadline` of 9 is to the right of Task 1 with `deadline` of 7). To find the highest-priority task with a given `core_cookie`, the scheduler explores `core_tree` as a binary search tree. If a node with the desired `core_cookie` is found, the scheduler then iterates through the left-side children to discover the highest-priority task with this `core_cookie`.

Note that core_tree is an rb_root and not an rb_root_cached. There is no benefit to caching the leftmost node of core_tree because there is no reason to believe the corresponding task would have the desired core_cookie. ▲

Sequence numbers. In Core Sched. 1, it was mentioned that the value of core_pick may no longer be valid when a CPU calls __schedule(). core_pick for a runqueue is invalidated whenever a task is enqueued or dequeued from any runqueue of a CPU in the same core. Validity is determined via observing three sequence numbers:

- core->core_task_seq, which is incremented whenever any runqueue in the core enqueues or dequeues a task and whenever new core_pick tasks are chosen;
- core->core_pick_seq, which is set to core->core_task_seq's value after a rescheduling CPU sets core_pick for the runqueues in the core;
- and core_sched_seq, which is set to core->core_pick_seq whenever a runqueue schedules its suggested core_pick task.

core_pick is valid if core->core_task_seq and core->core_pick_seq are equal and core->core_pick_seq and core->core_sched_seq are unequal. Agreement between core_task_seq and core_pick_seq indicates that the set of tasks available to be scheduled on the CPUs in the core has not changed since the core_pick tasks were computed. Otherwise, if a new highest-priority task was enqueued on a runqueue in the core, the core_cookie for the core must be set to this new task's core_cookie, necessitating new core_pick selections that match said core_cookie. Disagreement between core_pick_seq and core_sched_seq indicates that the current core_pick suggestion has not already been scheduled.

▼ **Example 4.15.** Consider a core with two CPUs as illustrated in Figure 4.18a. The rq of CPU 0 is the core runqueue, thus, both rq's core pointers point to the rq of CPU 0. Initially, both CPUs schedule the tasks with earliest deadlines on their respective rqs. This is Task 1 on CPU 0 and Task 3 on CPU 1. This is reflected in the curr pointers in both rqs. Both of these tasks have a core_cookie of 1. This matches the core_cookie in rq 0, the core runqueue.

Suppose Task 2 wakes on CPU 1, as illustrated in Figure 4.18b. Because a task was enqueued onto a rq in the core, core->core_task_seq is incremented from 10 to 11.

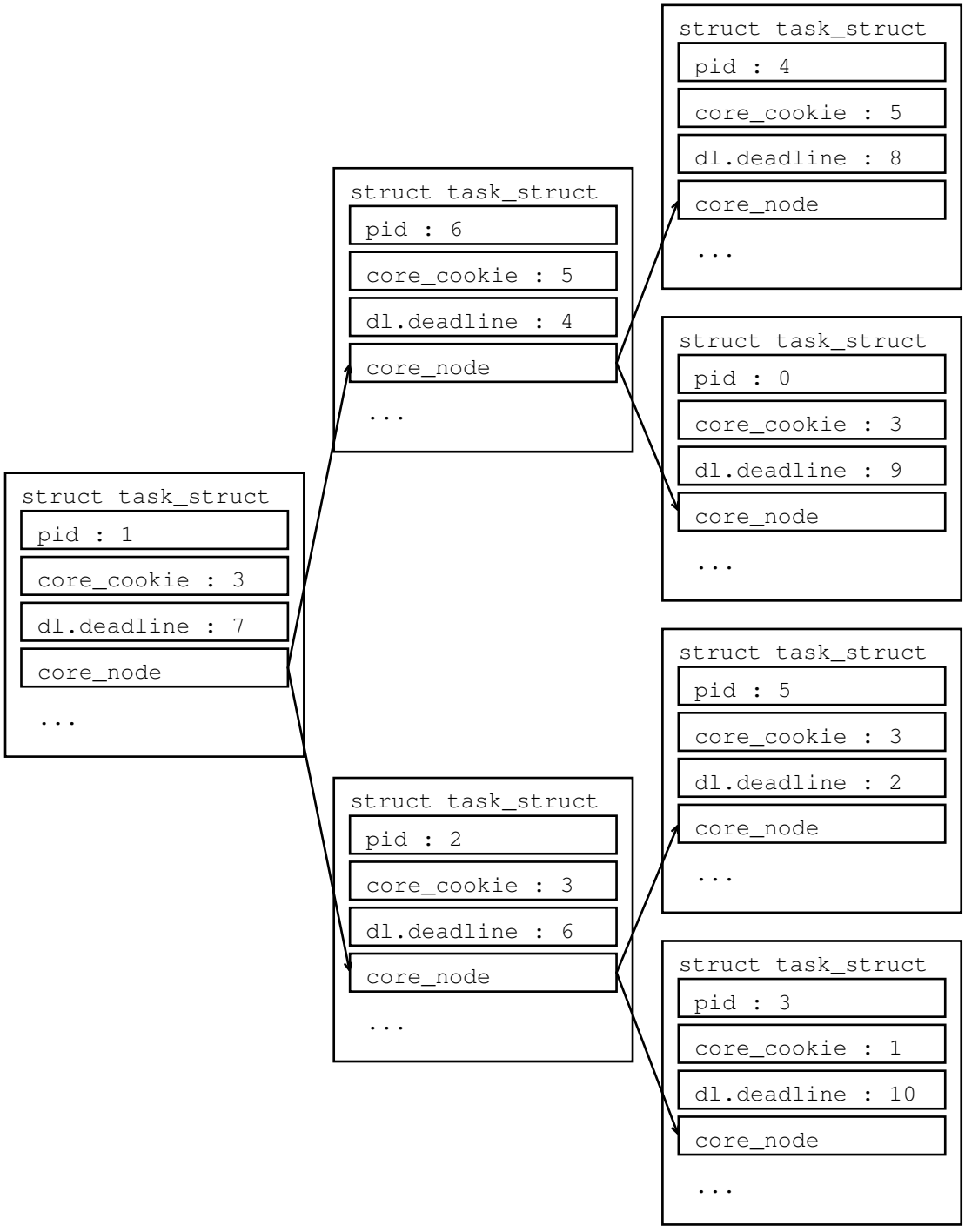


Figure 4.17: `core_tree` example.

CPU 1 reschedules due to the waking of Task 2. In Core Sched. 1 of core scheduling, CPU 1 decides whether or not to schedule its `rq`'s suggested task, which is pointed to by `core_pick` (which happens to be `NULL`). CPU 1 does not schedule its `core_pick` task because it observes that `core->core_task_seq` (11) and `core->core_pick_seq` (10) are unequal.

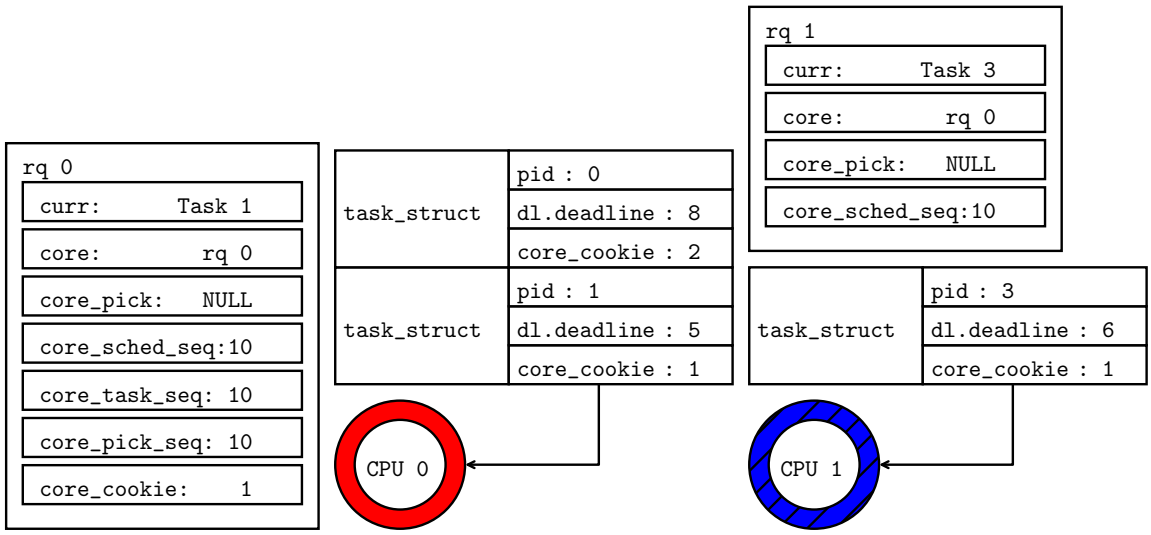
CPU 1 begins Core Sched. 2, which computes the new `core_cookie` for the core. The highest priority task on any `rq` in the core is identified. This is Task 2 on `rq` 1. CPU 1 sets `core->core_cookie` to Task 2's `core_cookie`, which is 2. CPU 1 also increments `core->core_task_seq` from 11 to 12. (Why `core_task_seq` is incremented here will be illustrated later in Example 4.16.) The state of the system is as illustrated in Figure 4.18c.

CPU 1 begins Core Sched. 3, which selects `core_pick` tasks for each `rq` in the core. On each of these `rq`s, `core_pick` is set to the highest-priority task on said `rq` that has `core_cookie` equal to `core->core_cookie` (2). This is Task 0 on `rq` 0 and Task 2 on `rq` 1. CPU 1 sets `core->core_pick_seq` to `core->core_task_seq` (12). The state of the system is as illustrated in Figure 4.18d.

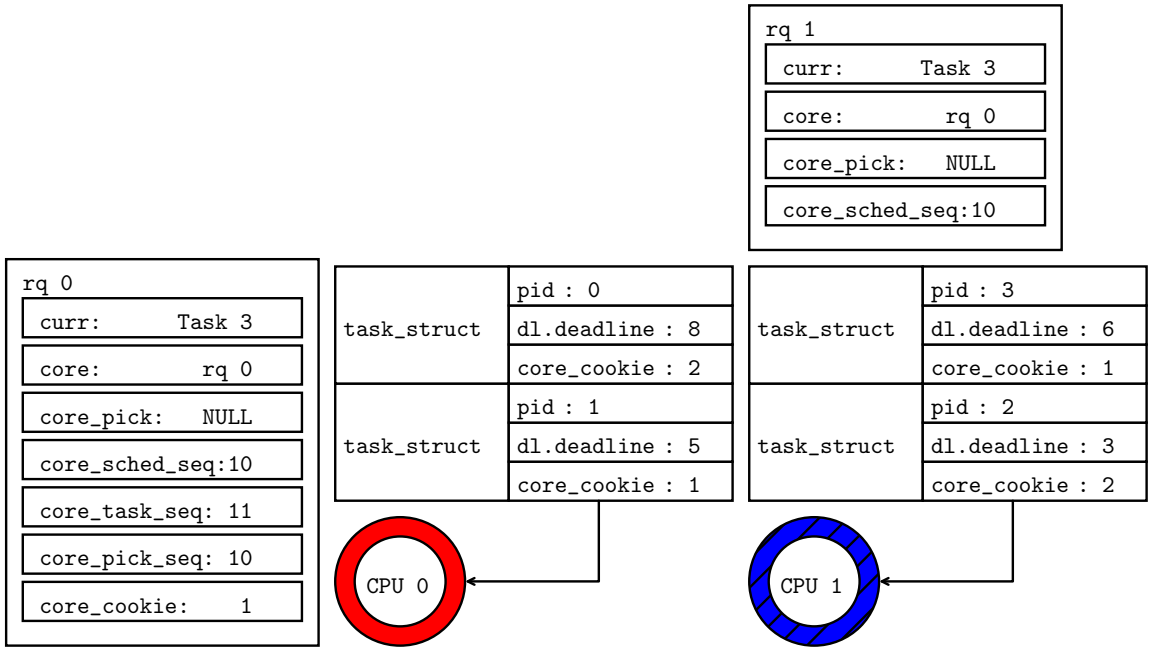
CPU 1 completes its rescheduling by scheduling its `core_pick` task (Task 2). This sets `curr` to Task 2 and `core_pick` to `NULL` on `rq` 1. `core_sched_seq` is also set to `core->core_pick_seq` (12). The state of the system is as illustrated in Figure 4.18e.

CPU 1 alerts CPU 0 to reschedule. When CPU 0 executes Core Sched. 1, it observes that `core->core_task_seq` is equal to `core->core_pick_seq` (12). This indicates that the set of tasks (Tasks 0-3) on the `rq`s in the core have not changed since `core_pick` was set on `rq` 0. CPU 0 also observes that `core->core_pick_seq` is unequal to `core_sched_seq` on `rq` 0. This indicates that the `core_pick` value on `rq` 0 has not already been scheduled on CPU 0. Because of these two observations, CPU 0 schedules its `core_pick` task (Task 0), as suggested by CPU 1. This sets `curr` to Task 0 and `core_pick` to `NULL` on `rq` 0. The system is as illustrated in Figure 4.18f. Each CPU schedules the highest-priority task on its `rq` that has `core_cookie` matching `core->core_cookie`. ▲

The second increment of `core->core_task_seq` from 11 to 12 between Figures 4.18b and 4.18c may seem unnecessary at first glance. The following example demonstrates how avoiding this increment can result in redundant rescheduling.

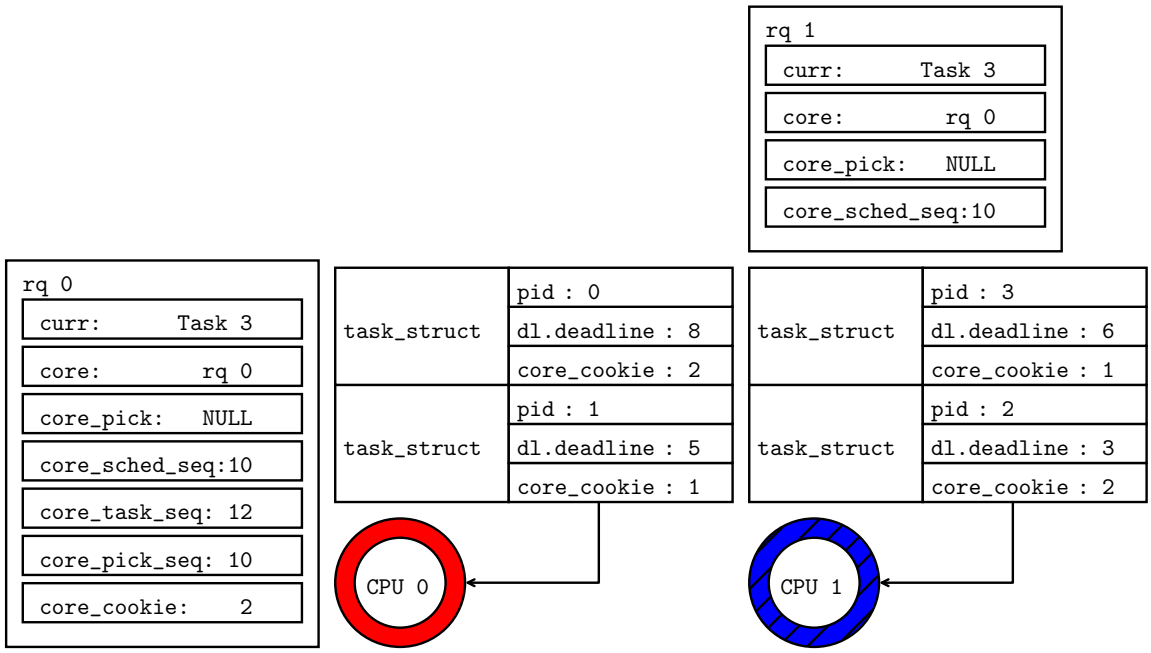


(a) Initial system.

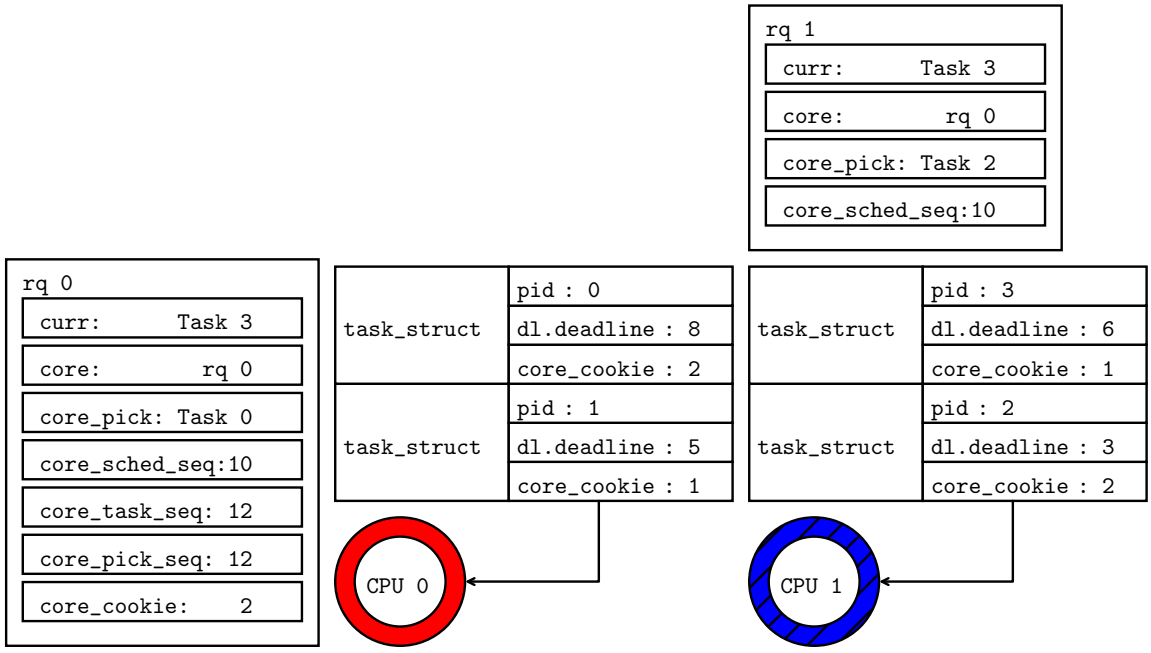


(b) Task 2 enqueued.

Figure 4.18: Core scheduling example.

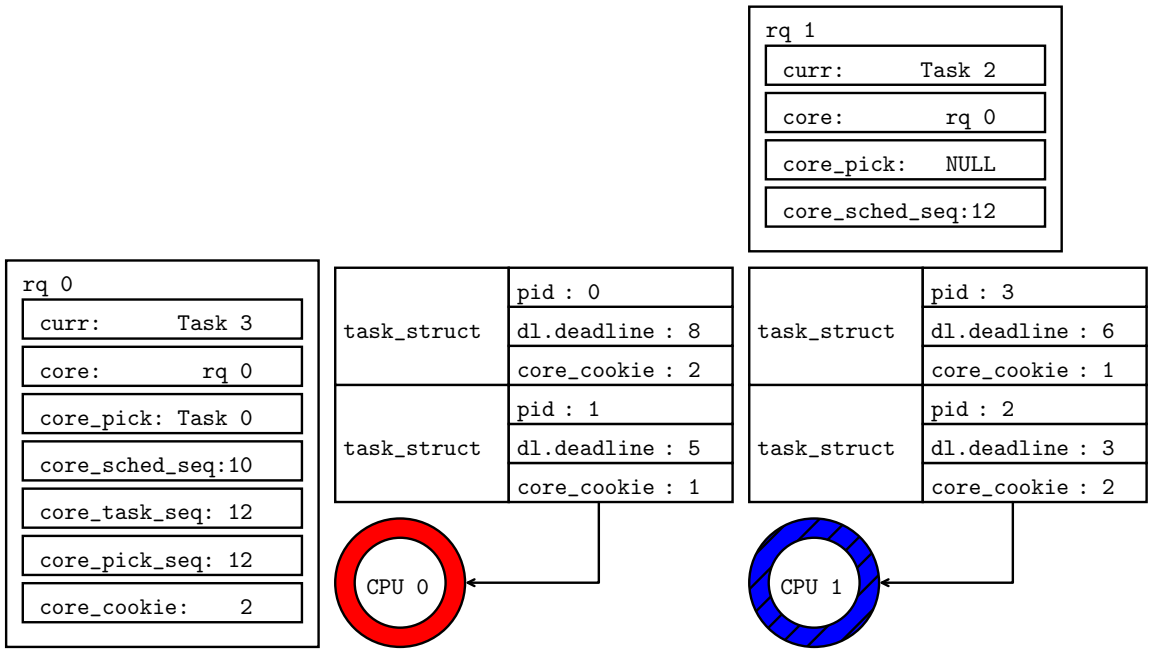


(c) New `core_cookie` selected.

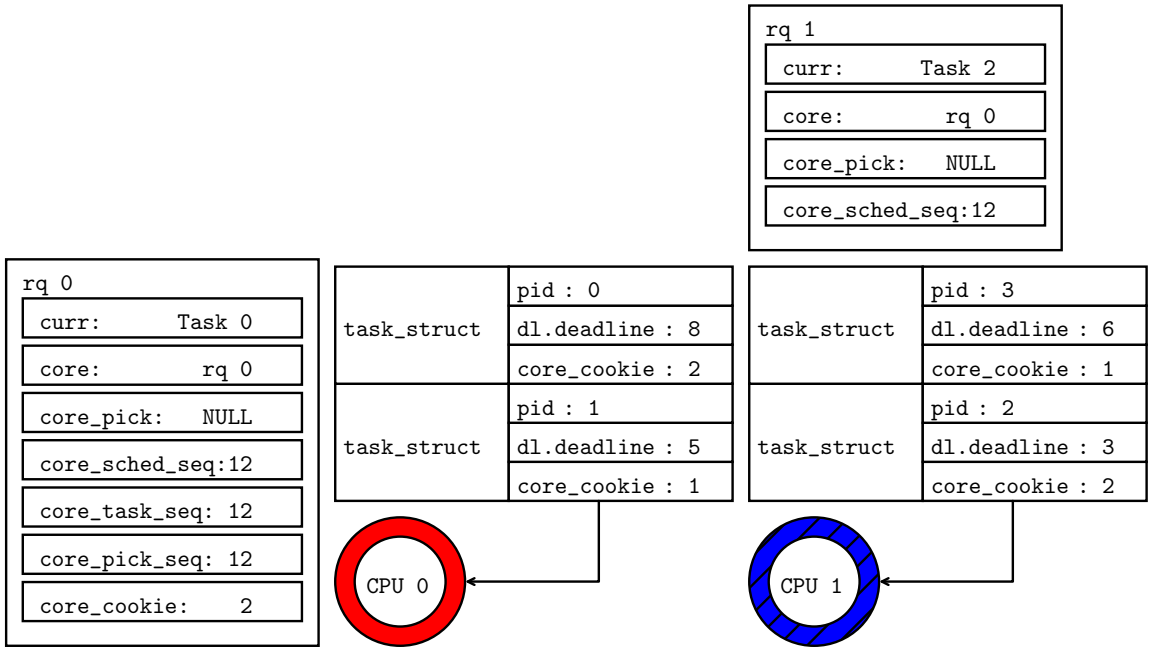


(d) New `core_picks` selected.

Figure 4.18: Core scheduling example (continued).



(e) CPU 1 finishes rescheduling.



(f) CPU 0 reschedules.

Figure 4.18: Core scheduling example (continued).

▼ **Example 4.16.** This example continues from the system in Example 4.15 and Figure 4.18f. Suppose that after being scheduled on CPU 1 for some duration, Task 2 exhausts its `runtime` or yields such that it bypasses being throttled. Task 2 increases its `deadline` from 3 to 11. Because throttling was bypassed, Task 2 is not dequeued from `rq` 1. Thus, `core->core_task_seq` is not incremented due to a task being enqueued or dequeued. Because Task 2 no longer has the earliest deadline among the tasks on `rq` 1, CPU 1 reschedules. The state of the system is as illustrated in Figure 4.19a.

When CPU 1 executes Core Sched. 1, it observes that `core->core_pick_seq` and `core_sched_seq` on `rq` 1 are equal. This indicates that `core_pick` on `rq` 1 is no longer valid.

CPU 1 executes Core Sched. 2, in which a new `core->core_cookie` value is selected. The highest-priority task on any `rq` in the core is Task 1 with `core_cookie` of 1. `core->core_cookie` is set to 1. The state of the system is illustrated in Figure 4.19b.

In the actual implementation, CPU 1 would increment `core->core_task_seq` from 12 to 13, as in between Figures 4.18b and 4.18c in Example 4.15. Consider the hypothetical where `core->core_task_seq` is not incremented and remains at 12.

CPU 1 executes Core Sched. 3, in which new `core_pick` tasks are selected. `core_pick` is set to Task 1 on `rq` 0 and Task 3 on `rq` 1. CPU 1 sets `core->core_pick_seq` to `core->core_task_seq`; however, because `core_task_seq` was not incremented, both `core_pick_seq` and `core_task_seq` remain at 12. The state of the system is illustrated in Figure 4.19c.

CPU 1 schedules Task 3, setting, on `rq` 1, `curr` to Task 3, `core_pick` to `NULL`, and `core_sched_seq` to 12. CPU 1 completes its rescheduling and alerts CPU 0 to reschedule. The state of the system is illustrated in Figure 4.19d.

CPU 0 executes Core Sched. 1. Ideally, CPU 0 should schedule the `core_pick` task set on `rq` 0 by CPU 1. Note that in Figure 4.19d, `core->core_pick_seq` equals `core_sched_seq` on `rq` 0. This is because `core_pick_seq` was unchanged when it was set to `core->core_task_seq` (in Figure 4.19c), which itself was previously not incremented (in Figure 4.19b). Because CPU 0 observes that `core->core_pick_seq` and `core_sched_seq` are equal, CPU 0 does not schedule its `core_pick` task, instead continuing on to Core Sched. 2.

After CPU 0 selects new `core_pick` tasks in Core Sched. 3, CPU 0 will alert CPU 1 to schedule its `core_pick` task. When CPU 1 reschedules, it will observe that `core->core_pick_seq` equals `core_sched_seq` on `rq` 1, causing CPU 1 to also continue on to Core Sched. 2. CPU 1 will alert

CPU 0 to reschedule in Core Sched. 3. The two CPUs will continuously force each other to reschedule until some task is enqueued or dequeued from `rq` 0 or 1, finally incrementing `core->core_task_seq`. ▲

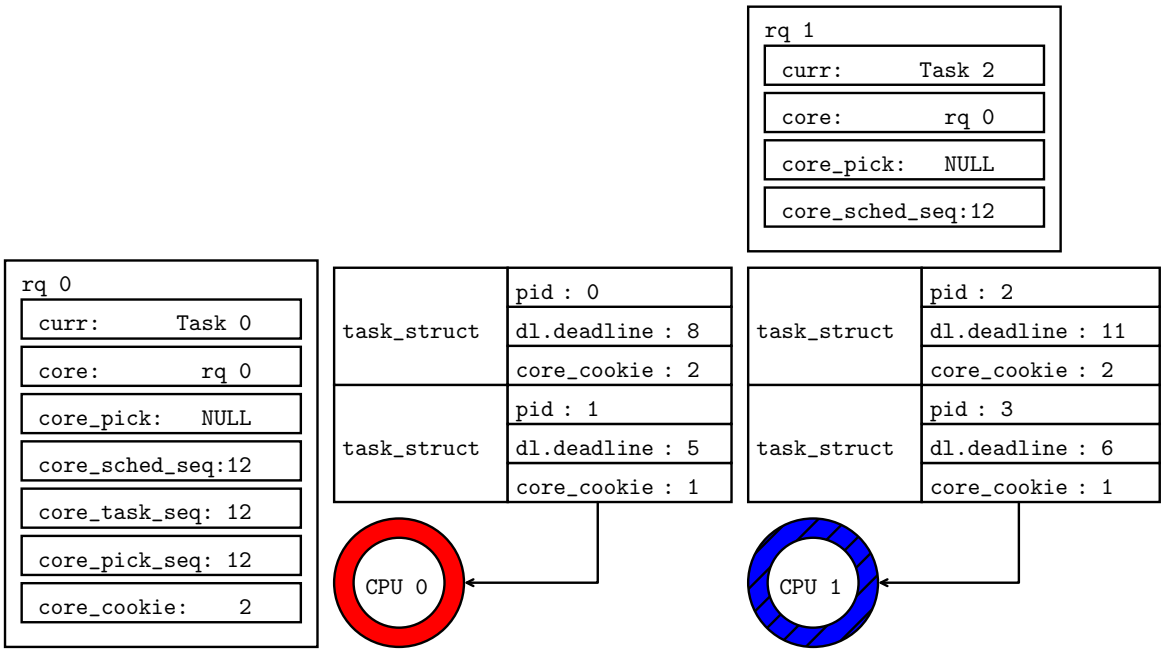
Example 4.17 demonstrates how incrementing `core->core_task_seq` whenever new `core_pick` tasks are selected avoids the redundant rescheduling shown in Example 4.16.

▼ **Example 4.17.** This example continues from the system illustrated in Figure 4.19a. CPU 1 increments `core->core_task_seq` from 12 to 13 in Figure 4.20a. When CPU 1 selects `core_pick` tasks for `rq`s 0 and 1, it sets `core->core_pick_seq` to `core->core_task_seq`, which is 13. This is reflected in Figure 4.20b. CPU 1 sets `core_sched_seq` on `rq` 1 to 13 when it finishes rescheduling (Figure 4.20c). When CPU 0 is alerted to reschedule by CPU 1, it observes that `core_sched_seq` on `rq` 0 (12) is unequal to `core->core_pick_seq` (13). Thus, CPU 0 schedules the `core_pick` task (Task 1) selected previously by CPU 1. CPU 0 then sets `core_sched_seq` to 13, as illustrated in Figure 4.20d. ▲

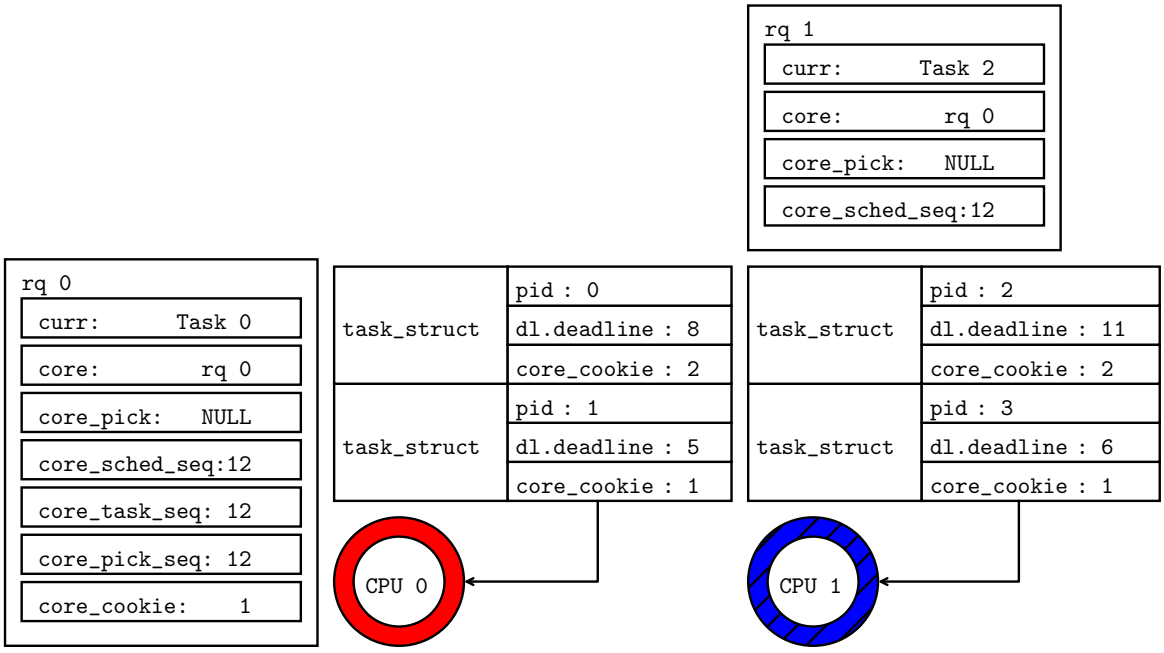
Balancing under core scheduling. CPUs whose runqueues are without any tasks matching the core's `core_cookie` are forced to schedule the `idle` task. CPUs forced to idle in this way attempt to pull tasks with matching `core_cookie` from other CPUs. As in `pull_dl_task()`, this pull iterates over the other CPUs. Note that this pull differs from those done by `pull_dl_task()` in that iteration terminates as soon as a single task is migrated. `pull_dl_task()`, in comparison, iterates over all CPUs in the `root_domain` in case a higher-priority task may be migrated from a CPU covered by a later iteration. Thus, core scheduling may leave higher-priority tasks unscheduled even when they could preempt lower-priority tasks with the same `core_cookie`.

4.5 Chapter Summary

In this chapter, we reviewed the `SCHED_DEADLINE` implementation as of kernel version 6.7. We discussed how the shared scheduling infrastructure calls `sched_class` functions in order to implement policies such as `SCHED_DEADLINE`. We have also briefly discussed features of the scheduler such as support for affinities, asymmetric capacities, priority inheritance, GRUB, DVFS, and core scheduling.

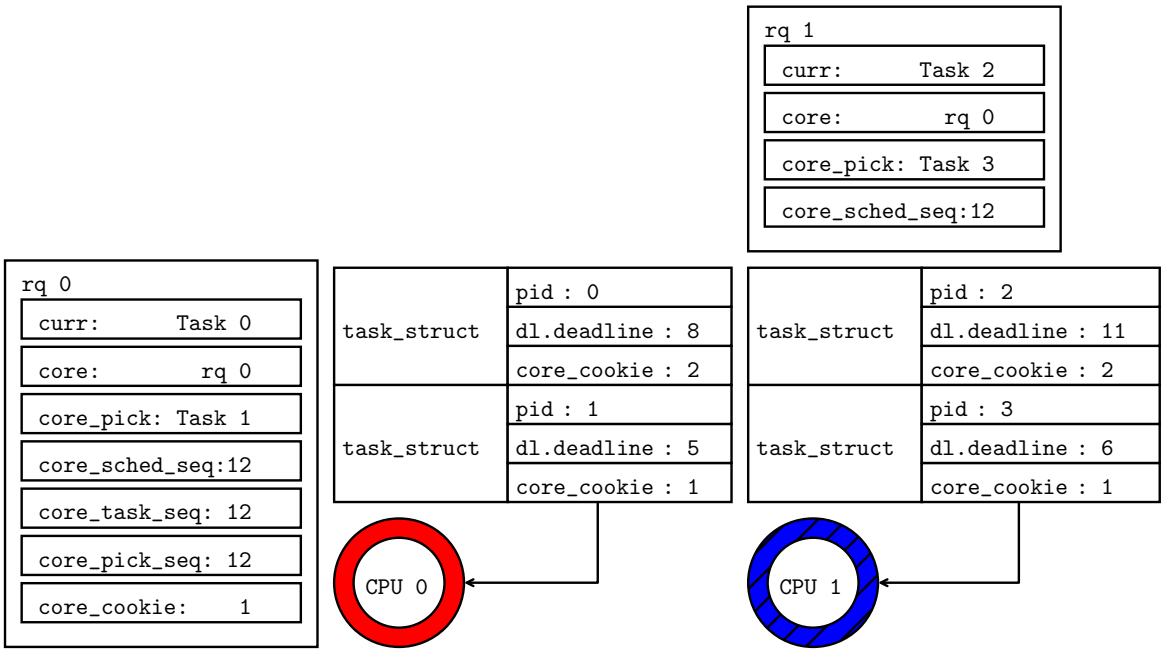


(a) Task 2 yields.

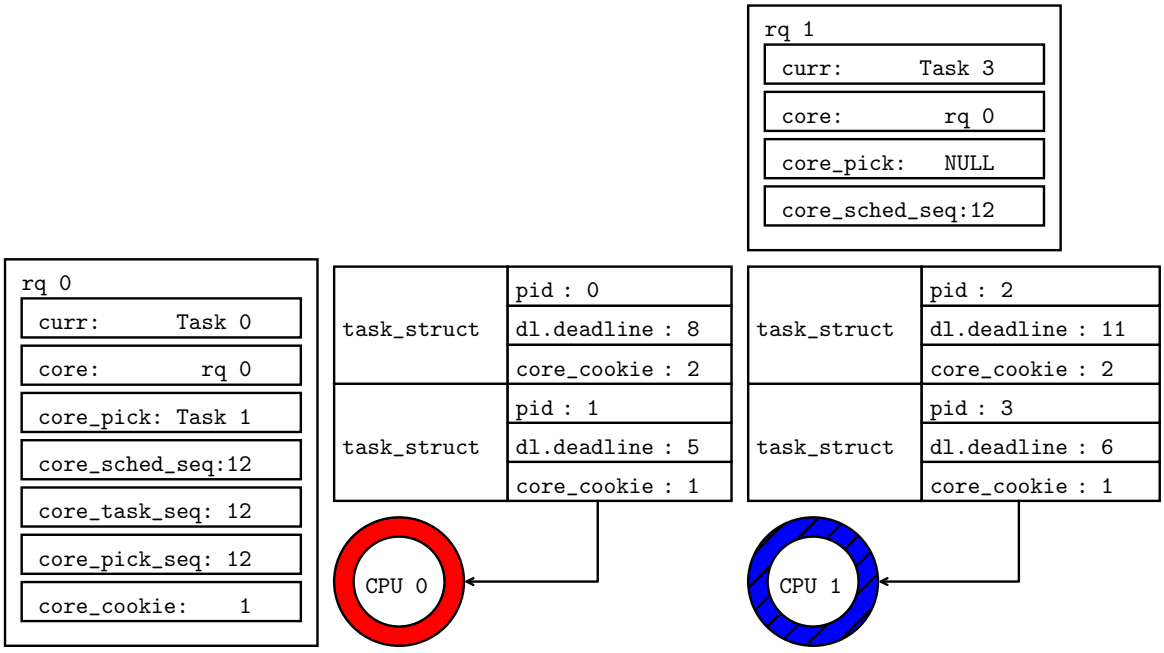


(b) New `core_cookie` selected.

Figure 4.19: Consequence of not incrementing `core_task_seq`.

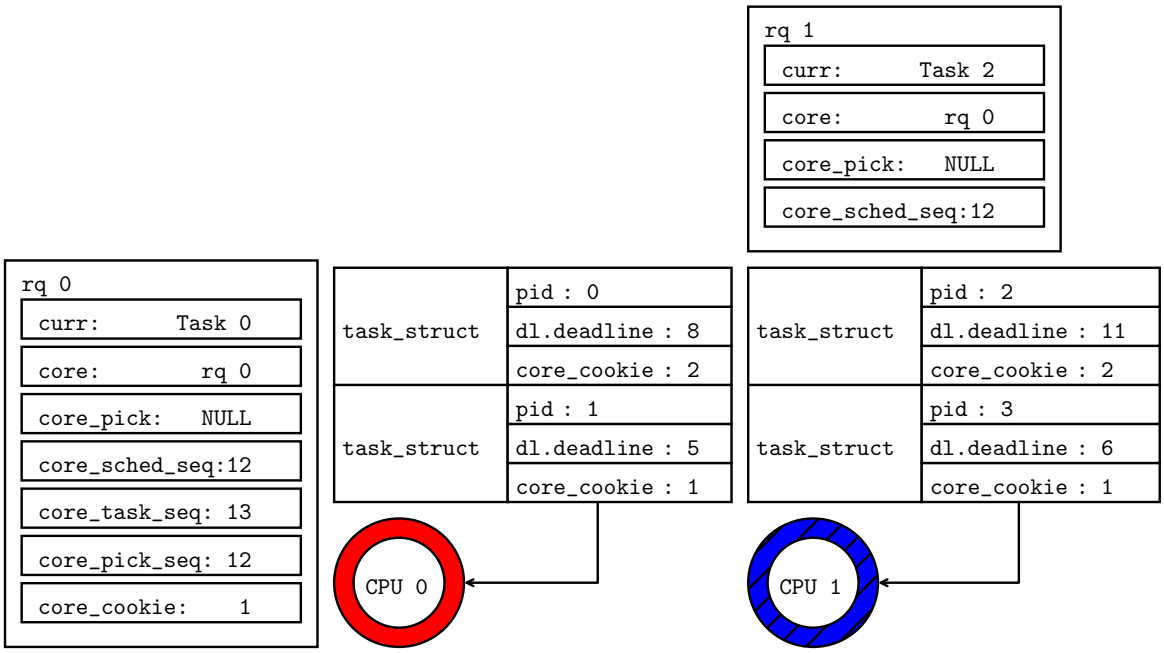


(c) New `core_picks` selected.

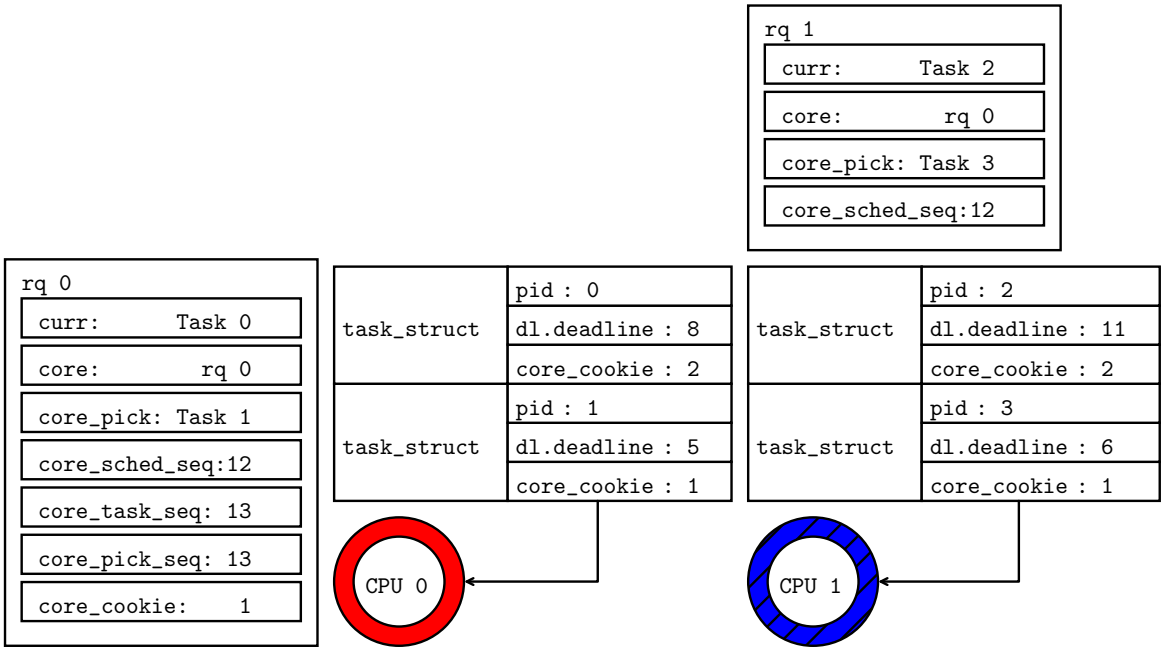


(d) CPU 1 finishes rescheduling.

Figure 4.19: Consequence of not incrementing `core_task_seq` (continued).

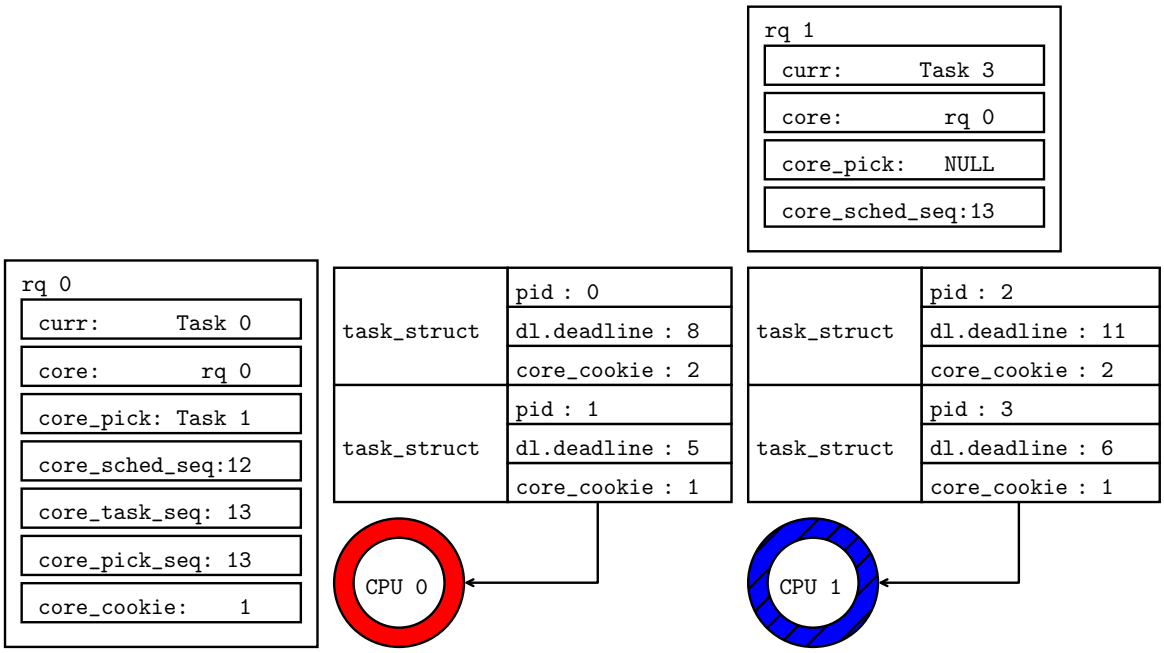


(a) New `core_cookie` selected.

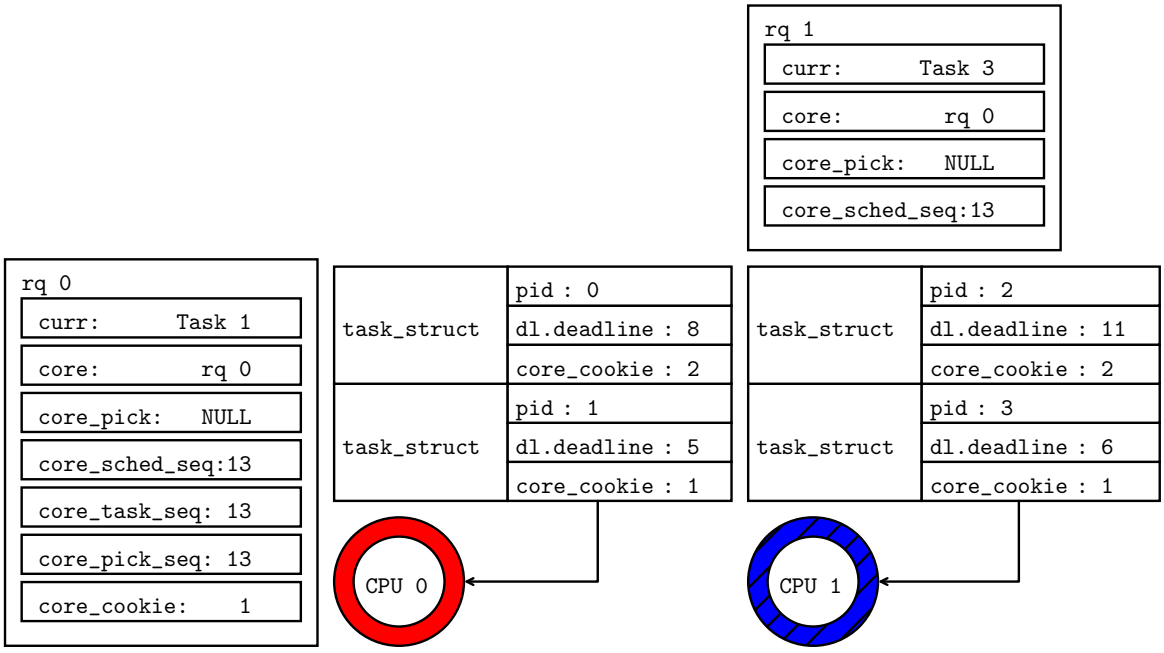


(b) New `core_picks` selected.

Figure 4.20: Correctly incremented `core_task_seq`.



(c) CPU 1 finishes rescheduling.



(d) CPU 0 reschedules.

Figure 4.20: Correctly incremented `core_task_seq` (continued).

CHAPTER 5: MODIFYING SCHED_DEADLINE [1]

This chapter presents two SCHED_DEADLINE patches aimed at restoring response-time bounds under the ACS for special cases of heterogeneous multiprocessors. Both patches are available online (Tang, a,b).

5.1 Version Differences

We briefly discuss differences between the kernels targeted by our patches and kernel 6.7, which was discussed in Chapter 4. Our first patch, which will be discussed in Section 5.2, targets kernel 5.4, which was the most recent long-term support kernel when this patch was first written. In the 5.4 kernel, the ACS and SCHED_DEADLINE migration logic do not take asymmetric capacities into account. This kernel also predates Linux scheduling features such as disabling migration and core scheduling.

Our second patch, which will be discussed in Section 5.3, targets a fork of the Linux kernel (Hardkernel) made by Hardkernel, the creators of the ODROID series of board computers. We use this fork because it supports the ODROID-XU4, the ARM big.LITTLE multiprocessor upon which we evaluate our patch. The branch we target is based on kernel version 6.1. From the perspective of SCHED_DEADLINE, kernels 6.1 and 6.7 are practically the same outside of some minor bug fixes and code refactoring.

Generally, though the organization of the code has since changed, the behavior of SCHED_DEADLINE (with the exception of the aforementioned unsupported features in the 5.4 kernel) in both earlier kernels is basically consistent with the discussion in Chapter 4.

5.2 IDENTICAL/SEMI-PARTITIONED

Our first patch (Tang, a) targets IDENTICAL/SEMI-PARTITIONED systems, *i.e.*, systems where all CPUs have the same capacity and each task has affinity for either all CPUs or one CPU. We target IDENTI-

[1] Contents of this chapter previously appeared in the following paper:

Stephen Tang, James H. Anderson, and Luca Abeni. On the defectiveness of SCHED_DEADLINE w.r.t. tardiness and affinities, and a partial fix. In *2021 29th International Conference on Real-Time Networks and Systems*, pages 46–56, 2021a.

CAL/SEMI-PARTITIONED systems for two reasons. The first reason is that setting SEMI-PARTITIONED affinities is a straightforward method of decreasing migration overheads. The second reason is that, as we will show in this section, response-time bounds can be restored for `SCHED_DEADLINE` under IDENTICAL/SEMI-PARTITIONED with minimal changes to the existing implementation.

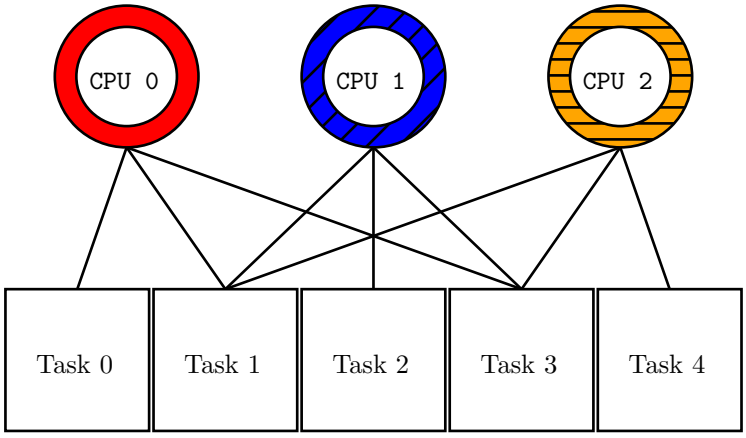
Our patch for IDENTICAL/SEMI-PARTITIONED attempts to change `SCHED_DEADLINE` to be closer to Strong-APA-EDF without requiring a major overhaul of the existing code. We describe our proposed patch in four subsections, each addressing a distinct aspect of the implementation: bypassing throttles, pushing to the latest CPU, the ACS, and dynamic affinities. We begin each subsection by reviewing how each aspect causes problems under IDENTICAL/SEMI-PARTITIONED. Afterwards, we explain how our patch modifies the implementation.

5.2.1 Bypassing Throttles

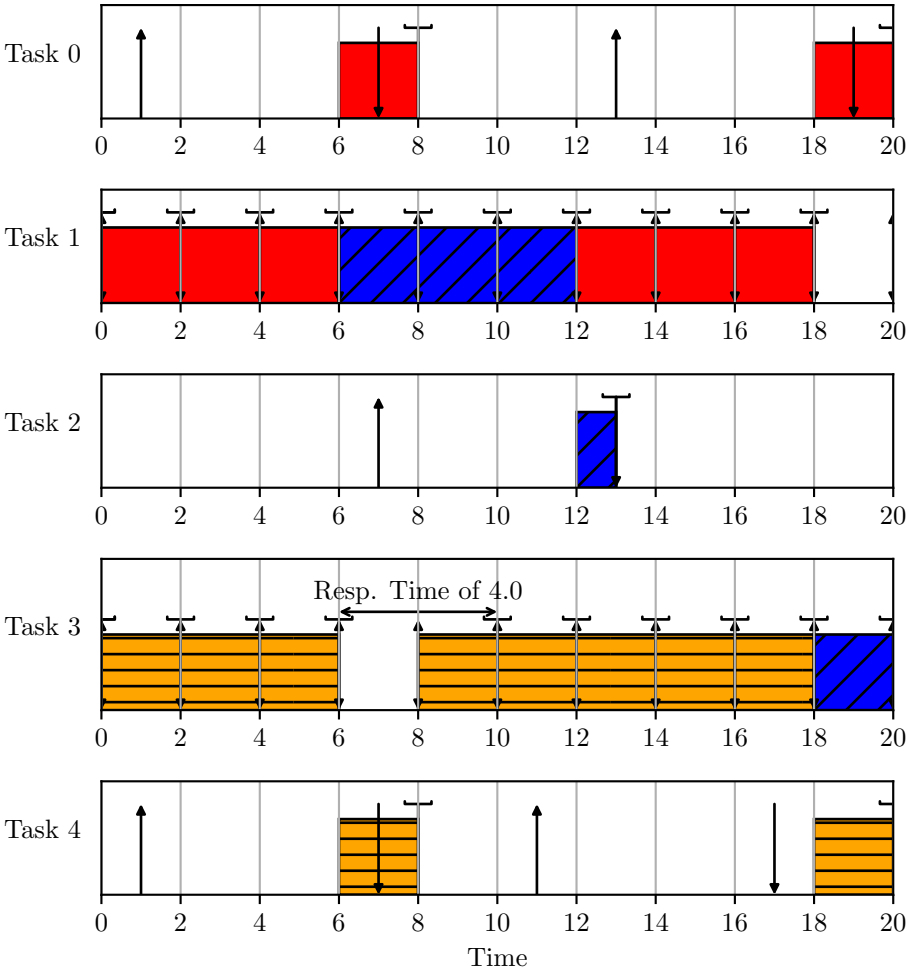
Problem. Recall from the discussion in Section 4.4.3 that in `SCHED_DEADLINE`, a task that exhausts its budget or calls `sched_yield` after its next replenishment time will remain eligible, bypassing the throttled state. If so, this task continues executing on the same CPU if it is not preempted by a different task on the same runqueue with an earlier deadline. Recall Example 3.1 in Section 3.4.1. This behavior can cause unbounded response times when partitioned tasks (*i.e.*, tasks with affinity for only a single CPU) exist.

▼ **Example 5.1.** Consider a `root_domain` with CPUs 0-2 with implicit-deadline Tasks 0-4. Tasks' affinities are illustrated in Figure 5.1a. Let $(\texttt{dl\_runtime}, \texttt{dl\_period})$ of Tasks 0 and 4 be $(2, 6)$, of Tasks 1 and 3 be $(2, 2)$, and of Task 2 be $(1, 6)$.

A schedule of this system is illustrated in Figure 5.1b. Tasks 1 and 3 are replenished every period. Initially, Task 1 and Task 3 execute on CPU 0 and CPU 2, respectively. Even though Tasks 0 and 4 become ready at time 1, because Tasks 1 and 3 bypass their throttled states due to exhausting their `runtime` at or past their deadlines (because these are implicit-deadline tasks, the next replenishment time is also the deadline), they do not migrate to the idle CPU 1. This continues until time 6 when fixed Tasks 0 and 4 preempt Tasks 1 and 3, respectively. The only other CPU available to both Task 1 and Task 3 is CPU 1 (Task 1 cannot preempt Task 4 on CPU 2 and Task 3 cannot preempt Task 0 on CPU 0), which they cannot both use. We assume the tie-break here favors Task 1 and it is scheduled, while Task 3 does not execute until time 8 when it resumes execution on CPU 2. Task 1 is also forced to migrate off of



(a) Example 5.1 affinity graph.



(b) Example 5.1 schedule.

Figure 5.1: Unbounded response times due to bypassing throttling.

CPU 1 by fixed Task 2 at time 12. This repeats at time 18, except here Task 3 is scheduled over Task 1 because it is tardy by 2.0 time units due to not being scheduled over $[6, 8)$.

Observe that this schedule is identical to the beginning of the schedule of the system in Example 3.1 and Figure 3.4 in Section 3.4.1 (though the tasks in Example 3.1 have different affinities than in this example, the migrations that are taken are the same). As in Figure 3.4, the schedule in Figure 5.1b can be repeated to yield unbounded response times. ▲

Patch. Example 5.1 would not have unbounded response times if successive jobs of Task 1 and Task 3 would migrate when a free CPU or CPU scheduling a task with a later deadline is available, thereby allowing partitioned tasks to execute. For example, if Task 1 had migrated to free CPU 1 at time 2 instead of time 6 in Figure 5.1b, then the first job of Task 0 would have been able to execute on CPU 0 over the interval $[2, 4)$. This would have freed CPU 0 over $[6, 8)$, allowing Task 3 to execute and preventing its increase in response time.

Recall that successive jobs of tardy tasks do not migrate because they bypass the throttled state, skipping the push that occurs in function `dl_task_timer()`, which is called when a task returns from being throttled. Our patch addresses this by removing the branch in which a task bypasses throttling. This causes successive jobs of tardy tasks that would have otherwise continued to execute on the same CPU to be pushed from that CPU. For example, at time 2 in Figure 5.1b, Task 1 completes its job. Under our patch, Task 1 would be throttled and immediately unthrottled because its next replenishment time is also at time 2 (instead of not being throttled at all, as in the original implementation). Due to calling `dl_task_timer()`, Task 1 would be pushed from CPU 0 to free CPU 1. This is the schedule described in the previous paragraph that reduces response times.

Note that `dl_task_timer()` will not push a scheduled task, as the scheduled task on a runqueue is removed from said runqueue's tree of pushable tasks (recall from Section 4.4.2.4 that `set_next_task_dl()` calls `dequeue_pushable_dl_task()`). This is problematic because throttled tasks may still be scheduled when `dl_task_timer()` executes because rescheduling is not instantaneous. To guarantee that a tardy task is pushed in our patch, `dl_task_timer()` may need to wait for the tardy task to be unscheduled. Unfortunately, it does not help to wait within `dl_task_timer()` for the relevant CPU to reschedule, as both `dl_task_timer()` and `__schedule()` are both required to hold the runqueue's lock to execute. In our patch, when `dl_task_timer()` executes and observes that the task to be pushed is

still scheduled, the callback releases the runqueue's lock and retries in the future, giving the relevant CPU the chance to reschedule. This is done by calling `hrtimer_forward()` within `dl_task_timer()` and returning `HRTIMER_RESTART`.

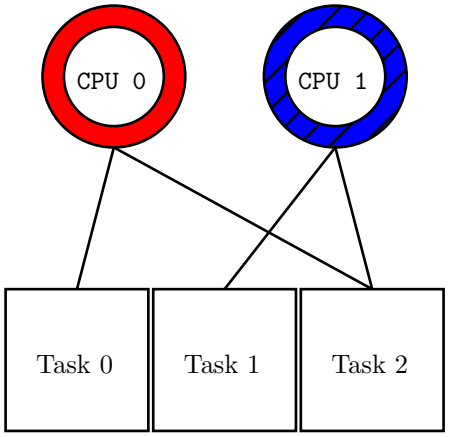
5.2.2 Pushing to the Latest CPU

Problem. The target CPU of a push is the CPU scheduling the task with the latest deadline if no CPUs in the `root_domain` are free of `SCHED_DEADLINE` tasks. In `SCHED_DEADLINE`, this latest CPU is identified via the `cpudl` heap. This heap orders CPUs by the earliest deadline of any task on the corresponding CPU's runqueue. Because a task being pushed is on its runqueue prior to being migrated, the deadline of the pushing CPU in the `cpudl` heap may be that of the task being pushed. This can result in priority inversions under SEMI-PARTITIONED.

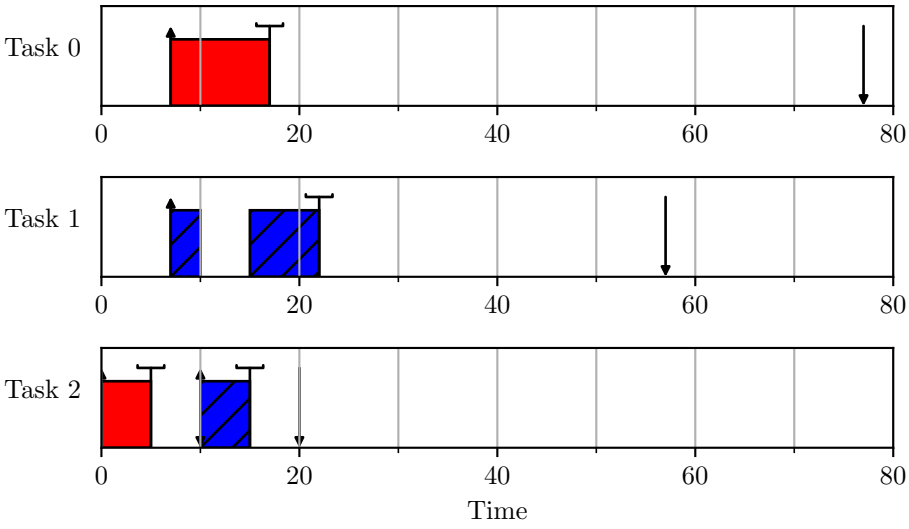
▼ **Example 5.2.** Consider a `root_domain` with CPUs 0 and 1 and implicit-deadline Tasks 0-2 with $(\texttt{dl\_runtime}, \texttt{dl\_period})$ of $(10, 70)$ for Task 0, $(10, 50)$ for Task 1, and $(5, 10)$ for Task 2. Tasks' affinities are illustrated in Figure 5.2a. A schedule for this system is presented in Figure 5.2b. Task 2 releases its first job at time 0 and executes on CPU 0 until time 5, at which point Task 2 is throttled. At time 7, both Task 0 and Task 1 release their first jobs and begin executing. At time 10, Task 2 is replenished. This results in Task 2 being placed back onto CPU 0's `dl_rq` (as CPU 0 was Task 2's last CPU) and CPU 0 attempting to push Task 2 (in `dl_task_timer()`).

Because, at the instant Task 2 is pushed, CPU 0 executes Task 0 with `deadline` 77 and CPU 1 executes Task 1 with `deadline` 57, Task 2 should remain on CPU 0 and preempt Task 0, whose `deadline` is later than that of Task 1. The actual behavior exhibited by `SCHED_DEADLINE` is that CPU 0 will have Task 2's `deadline` of 20 in the `cpudl` heap. This causes `SCHED_DEADLINE` to believe that CPU 1 with Task 1's `deadline` in the heap is the later CPU, resulting in Task 2 being pushed to CPU 1, preempting Task 1. ▲

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 (from the `dl_rq`, not the runqueue `rq`) any task in the process of being pushed before the `cpudl` is accessed to determine the target CPU (in function `find_later_rq()`). Dequeueing updates the `cpudl`, preventing the pushing CPU from being represented by the pushed task's `deadline` in the `cpudl`.



(a) Example 5.2 affinity graph.



(b) Example 5.2 schedule.

Figure 5.2: Pushes can cause priority inversions.

We enqueue a pushed task back onto the pushing CPU's `dl_rq` once `find_later_rq()` returns. Though this enqueue is redundant if a target CPU is successfully identified and the push does not fail due to the race conditions discussed in Section 4.4.2.2, as the task must then be dequeued once again to be enqueued onto the target CPU's `dl_rq`, enqueuing the task on its original `dl_rq` is necessary because `push_dl_task()` expects the task to be enqueued.

5.2.3 ACS

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

Patch. Our patch supports the creation of partitioned tasks by using `__sched_setaffinity()` to set a task's affinity to a single CPU in its `root_domain` prior to entering `SCHED_DEADLINE` (as will be discussed in the next subsection, `SCHED_DEADLINE` tasks are not allowed to change their affinities). We modify `__sched_setscheduler()` to not fail if a task has affinity for a single CPU.

Besides the condition in (4.1), we modify the ACS to also maintain,

$$\forall \pi_j \in \pi : \sum_{\tau_i:\ \alpha_i = \{\pi_j\}} u_i \leq \frac{\texttt{sched\_rt\_runtime\_us}}{\texttt{sched\_rt\_period\_us}}. \tag{5.1}$$

As in (4.1), π in (5.1) refers to the CPUs belonging to a single `root_domain`. Checks for (5.1) are added wherever `__dl_overflow()` is called and in `sched_dl_global_validate()` (recall from the discussion in Section 4.4.4 that these functions check that (4.2) will not be violated by a request). Similar to how the left-hand side of (4.1) is tracked in `total_bw` (recall that (4.2) is (4.1)'s equivalent in terms of `SCHED_DEADLINE` variables), the left-hand side of (5.1) is tracked in a new member `partitioned_bw` stored in each runqueue's `dl_rq`.

(5.1) is not checked in `dl_cpuset_cpumask_can_shrink()`, which is called when the set of CPUs in a `root_domain` changes. This is because the expected behavior under Linux when the CPUs in a `root_domain` are changed is that any affinity changes made with `__sched_setaffinity()` are lost. Thus, all partitioned `SCHED_DEADLINE` tasks in the `root_domain` will become global (for all CPUs in the `root_domain`), making (5.1) irrelevant. Because all partitioned tasks become global, `partitioned_bw` must be set to 0 for any CPUs in the `root_domain`.

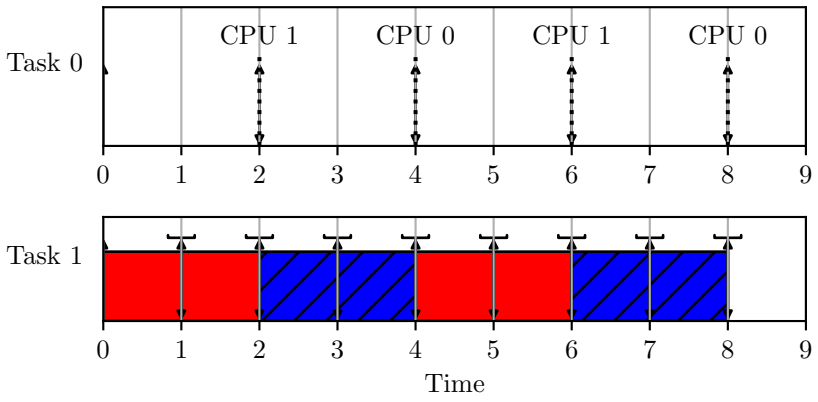


Figure 5.3: Dynamic affinities can starve tasks.

5.2.4 Dynamic Fine-Grained Affinities

Problem. We neglected to mention any checks made for (5.1) 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 `SCHED_DEADLINE` tasks to dynamically change their affinity can unpredictably change a tardy task's `deadline`.

▼ **Example 5.3.** This example corresponds with Figure 5.3. Consider a SEMI-PARTITIONED system on a `root_domain` with CPUs 0 and 1 and two implicit-deadline tasks Task 0 (initially partitioned on CPU 0) and Task 1 with affinity for both CPUs. The parameters (`dl_runtime`, `dl_period`) are $(1, 2)$ for Task 0 and $(1, 1)$ for Task 1. Both tasks enter the system at time 0.

Task 1, whose `deadline` is earlier than Task 0's `deadline`, executes on CPU 0 until preempted by Task 0 at time 2. Task 1 migrates to CPU 1 once preempted. However, Task 0 calls `__sched_setaffinity()` to change its affinity from being partitioned on CPU 0 to CPU 1. `__sched_setaffinity()` invokes the change pattern (see Section 4.3.5). When the change pattern enqueues Task 0 onto CPU 1's runqueue, `enqueue_task_dl()` is called with flag `ENQUEUE_RESTORE`. Because `enqueue_task_dl()` observes that `ENQUEUE_RESTORE` is set and that Task 0's `deadline` of 2 has passed, Task 0's `runtime` and `deadline` parameters are reset as if it entered `SCHED_DEADLINE` at time 2. Task 0's `deadline` is updated from 2 to 4. Thus, Task 0 does not have an early enough deadline to preempt Task 1, and so it continues to be unscheduled. Repeating this pattern of dynamic affinity requests can prevent Task 0 from executing indefinitely, as in Figure 5.3. ▲

Patch. We forbid SCHED_DEADLINE tasks from changing their affinities. We do this by modifying __sched_setaffinity() to automatically reject any requests for SCHED_DEADLINE tasks. If a user desires to change the affinity of a SCHED_DEADLINE task, the task must first leave SCHED_DEADLINE, change its affinity as a non-SCHED_DEADLINE task, and reenter SCHED_DEADLINE. This makes explicit to users that affinity changes will reset tasks' SCHED_DEADLINE parameters.

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 response times.

Altogether, our patch is fairly minor, modifying roughly 200 lines of code (for context, the main SCHED_DEADLINE file is roughly 3,000 lines of code).

5.2.5 Bounded Response Times

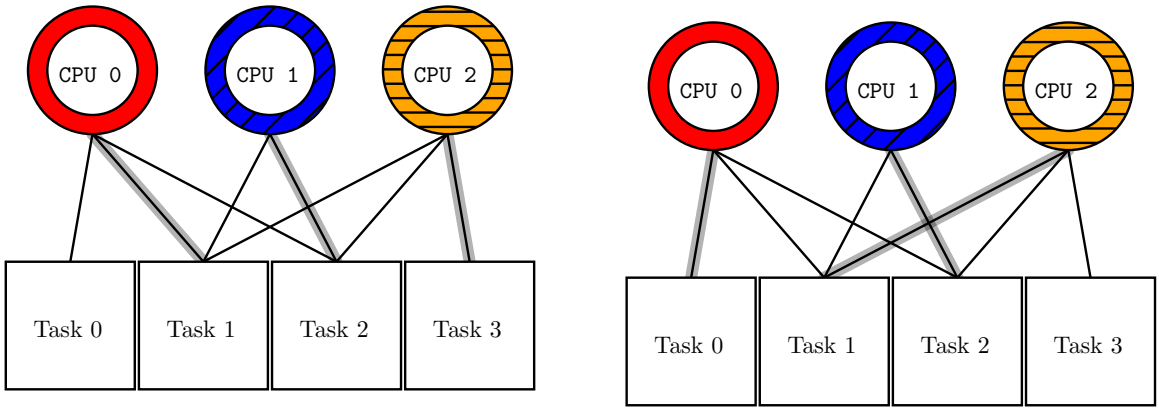
An objective of this patch is to guarantee that response times are bounded under the patched ACS.[2] We discuss at a high level why our changes to SCHED_DEADLINE result in bounded response times in this subsection. Formal details are presented in Appendix A.

Our patch is designed such that, assuming the relative order of deadlines is unchanged over a sufficiently long time interval, the configurations chosen by SCHED_DEADLINE in this interval approach a configuration that would be chosen by Strong-APA-EDF.

▼ **Example 5.4.** Consider four tasks and three CPUs such that affinites are as illustrated in Figure 5.4 and the relative order of deadline values from earliest to latest at the current time instant is Task 1, Task 2, Task 0, and Task 4. Suppose the configuration chosen at this time is as illustrated in Figure 5.4a. There is an alternating path of (Task 0, CPU 0, Task 1, CPU 2, Task 3) from a higher- to a lower-priority task.

Strong-APA-EDF would select a configuration that inverts the edges along this alternating path. Observe that inverting these edges involves a single migration of Task 1 from CPU 0 to CPU 2 (Figure 5.4b). Under our patch, Task 1 takes this migration once it is pushed in dl_task_timer(), which is never skipped due to bypassing throttling. Thus, the behavior under our patch matches that of

[2]This assumes an idealized version of SCHED_DEADLINE without features such as disabling migration, CPU hotplug, priority inheritance, GRUB, DVFS, core scheduling, *etc.*



(a) Alternating path of (Task 0, CPU 0, Task 1, CPU 2, Task 3).

(b) Migration of Task 1 inverts alternating path.

Figure 5.4: Alternating paths under SEMI-PARTITIONED.

Strong-APA-EDF after Task 1 exhausts its remaining `runtime` (due to Task 1 needing to be throttled for `dl_task_timer()` to be called). ▲

Under SEMI-PARTITIONED, for any alternating path from a higher-priority task to a lower-priority task or free CPU, there is always a shortest alternating path containing one global task such that inverting the edges of the shortest path results in the same tasks being scheduled as inverting the edges of the original path (*e.g.*, in Example 5.4, path (Task 0, CPU 0, Task 1, CPU 2, Task 3) is a shorter alternating path than (Task 0, CPU 0, Task 1, CPU 1, Task 2, CPU 2, Task 3), and inverting either path in Figure 5.4a results in Tasks 1, 2, and 3 being scheduled). Thus, there is no need to compute paths to achieve Strong-APA-EDF-like behavior. It is sufficient to migrate the global task in the shortest path to the latest CPU, which always occurs on replenishment under our patch.

As a consequence of waiting for a task's `runtime` to be exhausted before migrating to remove an alternating path, the response-time bounds derived under our patch are inflated from those derived for Strong-APA-EDF in Corollay 3.23. The details of this are presented in Appendix A.

5.2.6 Evaluation

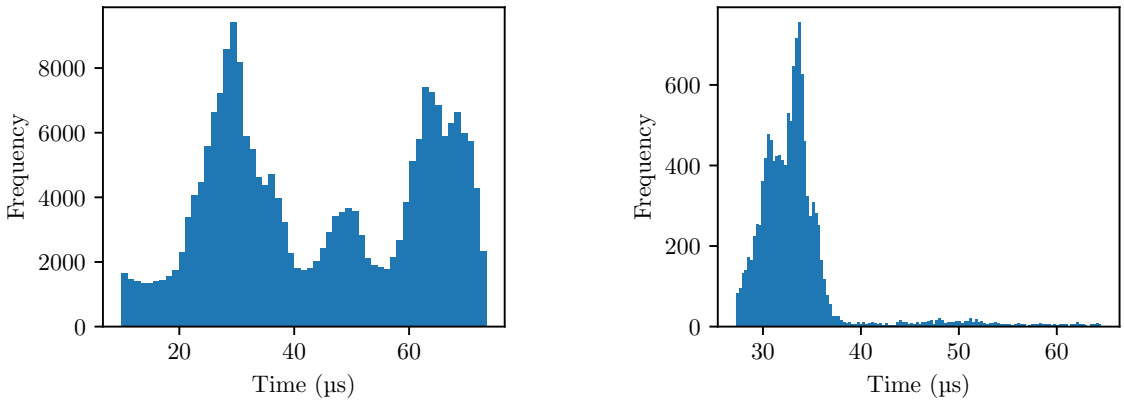
In this subsection, we evaluate the performance of our patched `SCHED_DEADLINE` kernel against the original implementation.

Experimental setup. Our experiments 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. Periodic workloads were generated for these experiments using `taskgen` (Emberson et al., 2010; Lelli, 2014) and `rt-app` (rt-app). Tasks with SEMI-PARTITIONED affinities were created by applying worst-fit packing to the task sets generated by `taskgen` and determining any unpacked tasks to have affinity for all eight CPUs.

We are interested in how our modifications to `SCHED_DEADLINE`'s migration code affect overheads. Changes to overheads are due to forcing tasks executing past their `deadline`s to be throttled, thereby requiring that such tasks wait for `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 waiting for `dl_task_timer()` (versus bypassing throttling), we inserted `ftrace` event tracepoints into our patched kernel that are triggered whenever a task executing past its `deadline` is forced into being throttled and when `dl_task_timer()` returns said task onto 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 migration code affect performance, we also measured the tardiness tasks experience scaled by their periods. Tardiness was measured instead of response times because `SCHED_DEADLINE` stores the `deadline`s of a task but not its arrival times.

`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 the ACS will not reject tasks due to potential rounding in `taskgen` in our patched kernel (the ACS must be disabled for SEMI-PARTITIONED scheduling in the unpatched kernel). 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 `SCHED_DEADLINE` implementation when considering the throttling of tardy tasks because this overhead is unique to our patched kernel. The distribution of sampled durations during which our patched `SCHED_DEADLINE` forced a task to be throttled when it would not have in the original implementation is presented in Figure 5.5. The average latency caused by a forced throttle was 44 µs for systems with 16 tasks and 34 µs for systems with 40 tasks. The



(a) $n = 16$. (b) $n = 40$.

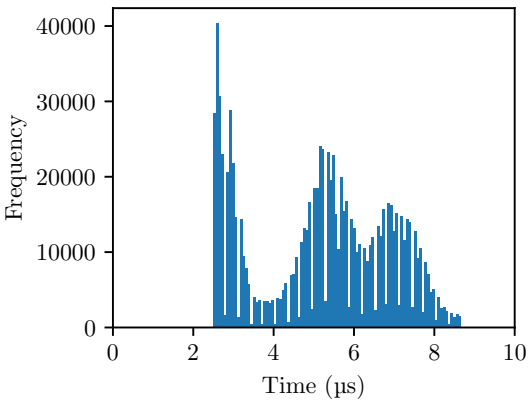
Figure 5.5: Forced Throttle Duration.

multimodal distribution of throttle times in Figure 5.5a is likely due to our usage of `hrtimer_forward()` within the `dl_task_timer()` callback to ensure that tardy tasks forced into throttling are unscheduled before `dl_task_timer()` attempts to push a task. The distance between peaks in the distribution roughly corresponds with the forwarding time of the `hrtimer`. As each usage of `hrtimer_forward()` 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. In Figure 5.5, higher competition for CPUs due to a higher number of tasks than in Figure 5.5a may cause the throttled task to always be unscheduled by the time `dl_task_timer()` is called, thereby removing the multimodal distribution.

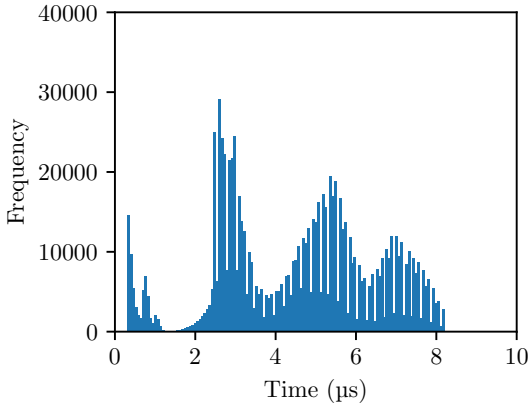
These latencies may not be acceptable for servers whose workloads require sub-ms response times. However, if the size of these latencies is being caused by waiting for the `hrtimer` as we suspect, an alternative method for forcing tasks that exhaust their `runtime`s to migrate that avoids using the `hrtimer` may be more practical.

Duration of pushes. The distributions of sampled push durations for both `SCHED_DEADLINE` and our patched kernel are presented in Figure 5.6. These distributions are multimodal because of the retry loop in `find_lock_later_rq()`, which is called by `push_dl_task()`. The effect of the added enqueue and dequeue operations on push overheads is minor, entailing a change of about 1 µs to the average duration of a push.

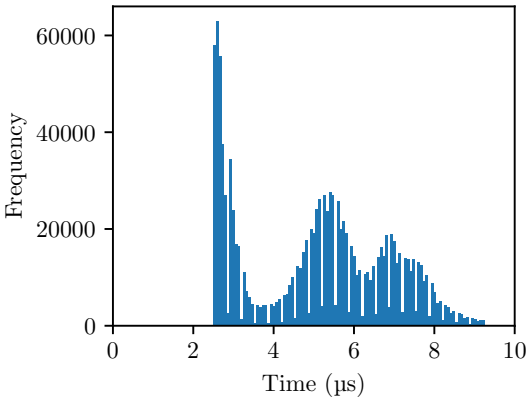
Tardiness. The distribution of samples of tasks' tardiness levels is presented in Figure 5.7. Average tardiness is negligibly lower under our patched kerenl compared to the original implementation.



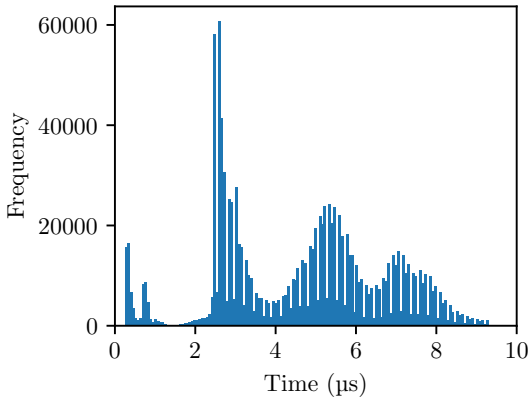
(a) Original ($n = 16$).



(b) Patched ($n = 16$).

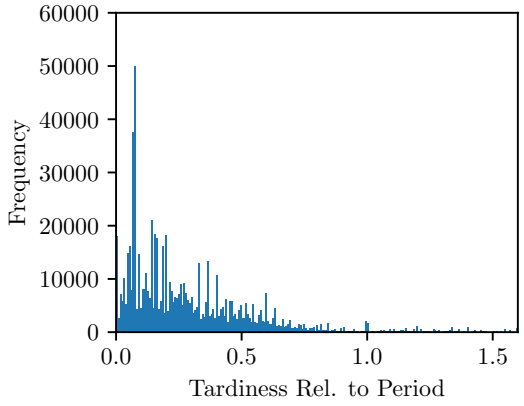


(c) Original ($n = 40$).

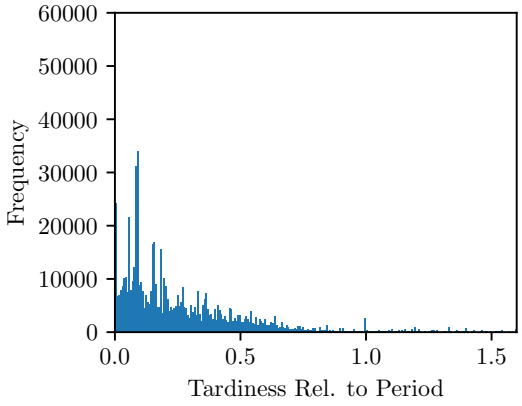


(d) Patched ($n = 40$).

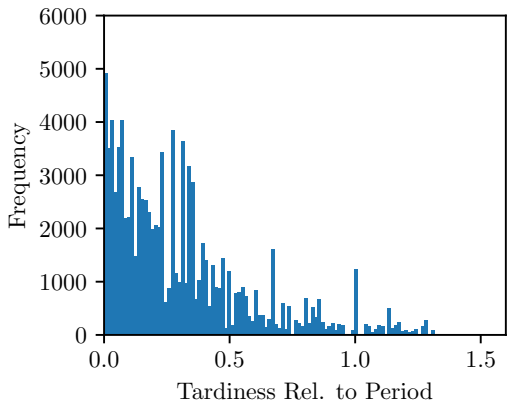
Figure 5.6: Push Durations.



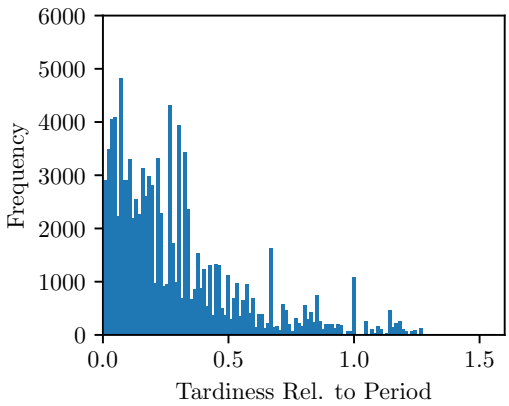
(a) Original ($n = 16$).



(b) Patched ($n = 16$).



(c) Original ($n = 40$).



(d) Patched ($n = 40$).

Figure 5.7: Tardiness.

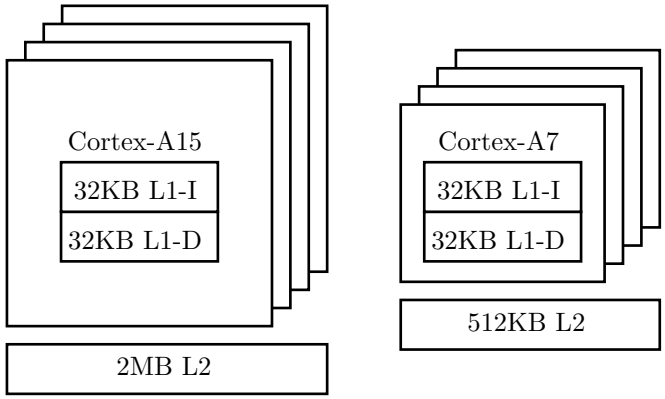


Figure 5.8: Samsung Exynos 5422.

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 Figures 5.5 and 5.6. The number of pushes performed by the system vastly outnumbers the number of instances where tardy tasks are forced to throttle.

5.3 UNIFORM/SEMI-CLUSTERED

Our patch for UNIFORM/SEMI-CLUSTERED (Tang, b) implements a special case of Unr-WC on a 2-type multiprocessor, *i.e.*, each processor has one of two speeds.

5.3.1 Hardware platform

This patch was implemented on the ODROID-XU4 (Roy and Bommakanti, 2017), which contains a Samsung Exynos 5422 multiprocessor, illustrated in Figure 5.8. This multiprocessor contains four big Cortex-A15 CPUs (2.0 GHz) and four LITTLE Cortex-A7 CPUs (1.4 GHz). Each set of four CPUs shares an L2 cache.

We impose SEMI-CLUSTERED affinities by requiring that any task has affinity for either all big CPUs, all LITTLE CPUs, or all CPUs. Having affinity for only big or only LITTLE CPUs is desirable because migrations between the same type of CPU result in fewer L2 cache misses. Some CPUs must have affinity for all CPUs in order to avoid bin-packing-related capacity loss.

5.3.2 Scheduler

Implementing Unr-WC directly requires solving its defined AP instance to compute an optimal configuration whenever a rescheduling is necessary. This is impractical due to the computational complexity of the AP (recall from Section 2.3 that solving an instance using the incremental method has $O(\max\{n, m\})$ time complexity) and the unpredictable migrations discussed in Section 3.5.1.1. This can be mitigated by implementing a simplified special case of Unr-WC, which we denote as Ufm-SC-EDF, for our considered multiprocessor model. Other examples of simplified special cases are Ufm-WC and Strong-APA-WC, which have lower computational complexity than Unr-WC and only migrate tasks on scheduling events (*i.e.*, job arrivals and completions).

Note that our proof that Ufm-SC-EDF is a special case of Unr-WC relies on the following two assumptions.

▷ **Constrained Deadlines Assumption.** All tasks are constrained-deadline tasks, *i.e.*, for any task $\tau_i \in \tau : D_i \leq T_i$. ◁

Recall from the discussion of the ACS in Section 4.4.4 that `SCHED_DEADLINE` already maintains the Constrained Deadlines Assumption.

▷ **No-Early-Releasing Assumption.** There is no early releasing, *i.e.*, for any job $\tau_{i,j}$, $rdy_{i,j} \geq a_{i,j}$. ◁

Note that priority inheritance (recall Section 4.4.7) is incompatible with the No-Early-Releasing Assumption, and should not be used in conjunction with our patch.

We will later use these assumptions to define priority points under Ufm-SC-EDF that mitigate the issues inherent to Unr-WC (of which Ufm-SC-EDF is a special case) discussed in Section 3.5.1.1.

Our definition of Ufm-SC-EDF relies on Definitions 5.1-5.8, presented below.

▽ **Definition 5.1.** The set of big CPUs is denoted π^{big} and the set of LITTLE CPUs is denoted π^{LIT}. The number of big and LITTLE CPUs are denoted as m^{big} and m^{LIT}, respectively.

The subset of tasks τ_i with affinity $\alpha_i = \pi^{\text{big}}$ is denoted τ^{big}, with affinity $\alpha_i = \pi^{\text{LIT}}$ is denoted τ^{LIT}, and with affinity $\alpha_i = \pi = \pi^{\text{big}} \cup \pi^{\text{LIT}}$ is denoted τ^{glob}.

At time t, the subset of active tasks in τ^{big}, τ^{LIT}, and τ^{glob} are denoted $\tau^{\text{big}}_{\text{act}}(t)$, $\tau^{\text{LIT}}_{\text{act}}(t)$, and $\tau^{\text{glob}}_{\text{act}}(t)$, respectively. △

It follows from our platform that $\pi = \pi^{\text{big}} \cup \pi^{\text{LIT}}$, $\tau = \tau^{\text{big}} \cup \tau^{\text{LIT}} \cup \tau^{\text{glob}}$, and, for any time t, $\tau_{\text{act}}(t) = \tau_{\text{act}}^{\text{big}}(t) \cup \tau_{\text{act}}^{\text{LIT}}(t) \cup \tau_{\text{act}}^{\text{glob}}(t)$.

▽ **Definition 5.2.** The speed of a big CPU $\pi_j \in \pi^{\text{big}}$ is 1.0. The speed of a LITTLE CPU $\pi_j \in \pi^{\text{LIT}}$ is denoted as $sp^{\text{L}} \in (0, 1.0)$. △

Definition 5.2 reflects that the maximum capacity of any CPU in Linux is 1.0 (recall Section 4.4.6).

In order to schedule an optimal configuration, it is sometimes necessary to migrate a running task $\tau_i \in \tau^{\text{glob}}$ from a LITTLE to a big CPU, even if no other task preempts this running task. Such migrations are analogous to those under Ufm-EDF where running tasks migrate to faster CPUs when the formerly higher-priority tasks running on these CPUs suspend. For our to-be-presented definition of Ufm-SC-EDF, it is convenient to represent such migrations as task τ_i being preempted by an *idle* task, defined below.

▽ **Definition 5.3.** The set of m *idle tasks*, which is disjoint from the set of real-time tasks τ, is denoted $\tau^{\text{idle}} \triangleq \{\tau_{n+1}, \tau_{n+2}, \ldots, \tau_{n+m}\}$. 'Scheduling' an idle task is symbolic of not scheduling any real-time task in τ. For each idle task $\tau_i \in \tau^{\text{idle}}$, the *deadline* of τ_i is defined as $d_i(t) \triangleq t + T_{[1]}$.

For each CPU $\pi_j \in \pi$, the affinity of idle task τ_{n+j} is defined as $\alpha_{n+j} \triangleq \{\pi_j\}$, *i.e.*, the idle task τ_{n+j} only has affinity for CPU π_j. △

While tasks of τ denote `SCHED_DEADLINE` tasks, tasks of τ^{idle} are named after the `idle` tasks on each `rq`. Recall that, in Linux, each CPU always schedules *some* task because the `idle` task on a `rq` is always ready. Tasks of τ^{idle} are similar in that every CPU π_j is guaranteed to schedule some task (either in τ or τ^{idle}) in an *augmented configuration*, formalized in Definitions 5.4 and 5.5 below.

▽ **Definition 5.4.** The *augmented* set of tasks is $\bar{\tau} \triangleq \tau \cup \tau^{\text{idle}}$. The augmented set of active tasks at time t is $\bar{\tau}_{\text{act}}(t) \triangleq \tau_{\text{act}}(t) \cup \tau^{\text{idle}}$. The augmented set of ready tasks at time t is $\bar{\tau}_{\text{rdy}}(t) \triangleq \tau_{\text{rdy}}(t) \cup \tau^{\text{idle}}$.△

For any time t, we have $\tau^{\text{idle}} \subseteq \bar{\tau}_{\text{rdy}}(t)$ because any idle task is always ready, *i.e.*, not scheduling a `SCHED_DEADLINE` task is always an option on any CPU.

▽ **Definition 5.5.** For a configuration $\mathbf{X}$, its *augmented configuration* $\bar{\mathbf{X}} \in \mathbb{R}^{(n+m)\times m}$ is $\mathbf{X}$ vertically concatenated with the $m \times m$ matrix such that $\forall \pi_j \in \pi : \big(\forall \tau_i \in \tau_{\text{rdy}}(t) : x_{i,j} = 0\big) \Rightarrow \bar{x}_{n+j,j} = 1$, *i.e.*, any CPU π_j not matched with a ready task in $\mathbf{X}$ is matched to its corresponding idle task τ_{n+j} in $\bar{\mathbf{X}}$.△

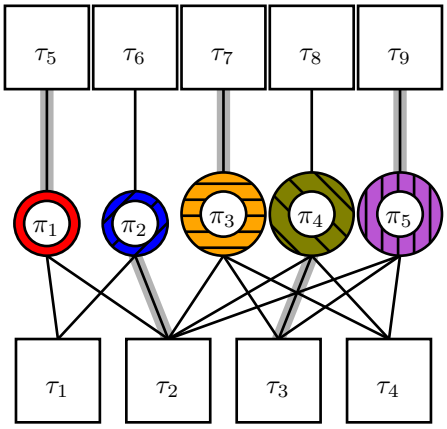


Figure 5.9: Augmented configuration $\bar{\mathbf{X}}^{(2)}$.

▼ **Example 5.5.** Recall the configuration $\mathbf{X}^{(2)}$ from Example 2.7 and illustrated in Figure 2.6b. Matrix

$$\bar{\mathbf{X}}^{(2)} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix},$$

illustrated in Figure 5.9, is the augmented configuration corresponding with $\mathbf{X}^{(2)}$. CPUs π_1, π_3, and π_5, which are unmatched in $\mathbf{X}^{(2)}$, are matched to idle tasks τ_5, τ_7, and τ_9 in $\bar{\mathbf{X}}^{(2)}$, respectively. ▲

It follows from Definition 5.5 that there is a one-to-one mapping between configurations and augmented configurations. Going forward, the absence or presence of an overline (*e.g.*, $\mathbf{X}$ and $\bar{\mathbf{X}}$) is used to denote a given configuration and its corresponding augmented configuration.

▽ **Definition 5.6.** The *deadline* of CPU π_j at time t under configuration $\bar{\mathbf{X}}$ is $d_j^{\text{CPU}}\big(\bar{\mathbf{X}}, t\big) \triangleq d_i(t)$, where i is such that $\bar{x}_{i,j} = 1$, *i.e.*, task $\tau_i \in \bar{\tau}_{\text{rdy}}(t)$ is matched to π_j in $\bar{\mathbf{X}}$. △

In words, $d_j^{\text{CPU}}\big(\bar{\mathbf{X}}, t\big)$ denotes the deadline at time t of the task matched with CPU π_j in configuration $\bar{\mathbf{X}}$. This task must exist because CPU π_j must be matched in $\bar{\mathbf{X}}$ with either a `SCHED_DEADLINE` task in τ or CPU π_j's corresponding idle task $\tau_{n+j} \in \tau^{\text{idle}}$.

▽ **Definition 5.7.** Let

$$d^{\text{big}}\big(\bar{\mathbf{X}}, t\big) \triangleq \max_{\pi_j \in \pi^{\text{big}}} \left\{ d_j^{\text{CPU}}\big(\bar{\mathbf{X}}, t\big) \right\}$$

and

$$d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big) \triangleq \max_{\pi_j \in \pi^{\text{LIT}}} \left\{ d_j^{\text{CPU}}\big(\bar{\mathbf{X}}, t\big) \right\}. \qquad \triangle$$

In words, $d^{\text{big}}\big(\bar{\mathbf{X}}, t\big)$ and $d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big)$ are the latest deadlines at time t of the tasks matched with big CPUs in $\bar{\mathbf{X}}$ and with LITTLE CPUs in $\bar{\mathbf{X}}$, respectively.

Each task is assigned a *weighted deadline* (to be defined in Definition 5.8) that depends on $d^{\text{big}}\big(\bar{\mathbf{X}}, t\big)$ and $d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big)$. The intuition behind weighted deadlines is to allow tasks in τ^{big}, which must execute exclusively on big CPUs, and tasks in τ^{LIT}, which must execute exclusively on LITTLE CPUs, to preempt a running task in τ^{glob} if doing so would cause said task in τ^{glob} to migrate to a more preferable CPU. For example, suppose that at a time t, a task $\tau_e \in \tau^{\text{glob}}$ is running on a LITTLE CPU while a big CPU runs an idle task and there is an unscheduled task $\tau_\ell \in \tau^{\text{LIT}}$ such that $d_e(t) < d_\ell(t)$, *i.e.*, task τ_e has an earlier `deadline` than task τ_ℓ. Even though τ_e has an earlier deadline, it is preferable for τ_ℓ to preempt τ_e because doing so schedules τ_ℓ, which was formerly unscheduled, and allows τ_e to migrate to a faster CPU. We allow τ_ℓ to preempt task τ_e by comparing task τ_ℓ's `deadline` against task τ_e's weighted deadline in the preemption code instead of task τ_e's `deadline`.

On the other hand, suppose that $\tau_e \in \tau^{\text{glob}}$ is scheduled on a big CPU while a LITTLE CPU schedules an idle task and $\tau_\ell \in \tau^{\text{big}}$ is unscheduled. Even if τ_e has an earlier `deadline` than τ_ℓ, if the `deadlines` of τ_e and τ_ℓ are 'close,' it is preferable that τ_e migrate to this LITTLE CPU to make room for τ_ℓ. Again, we allow τ_ℓ to preempt τ_e by comparing task τ_ℓ's `deadline` against task τ_e's weighted deadline.

Because only tasks in τ^{glob} have access to the CPUs in both π^{big} and π^{LIT} (and thus, can potentially be migrated to more preferable CPUs), only the tasks in τ^{glob} have weighted deadlines distinct from their

`deadlines`. How much later the weighted deadline of a task $\tau_e \in \tau^{\text{glob}}$ is from its `deadline` is determined by $d^{\text{big}}\big(\bar{\mathbf{X}}, t\big)$ if τ_e is scheduled on a LITTLE CPU in $\bar{\mathbf{X}}$ and $d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big)$ if τ_e is scheduled on a big CPU in $\bar{\mathbf{X}}$.

▽ **Definition 5.8.** Given configuration $\bar{\mathbf{X}}$, time t, and processor π_j, the *weighted deadline* of task τ_i is

$$\bar{d}_i\big(\bar{\mathbf{X}}, t\big) \triangleq \begin{cases} \big(1.0 - sp^{\text{L}}\big) \cdot d_i(t) + sp^{\text{L}} \cdot d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big) & \tau_i \in \tau^{\text{glob}},\ \bar{x}_{i,j} = 1,\ \pi_j \in \pi^{\text{big}}, \\ & \text{and } d_i(t) < d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big) \\ \frac{1.0}{sp^{\text{L}}} \cdot d^{\text{big}}\big(\bar{\mathbf{X}}, t\big) - \frac{1.0 - sp^{\text{L}}}{sp^{\text{L}}} \cdot d_i(t) & \tau_i \in \tau^{\text{glob}},\ \bar{x}_{i,j} = 1,\ \pi_j \in \pi^{\text{LIT}}, . \\ & \text{and } d_i(t) < d^{\text{big}}\big(\bar{\mathbf{X}}, t\big) \\ d_i(t) & \text{otherwise} \end{cases} \quad \triangle$$

Having covered Definitions 5.1-5.8, we now present the definition of Ufm-SC-EDF.

▷ **Ufm-SC-EDF.** At time t, a configuration $\bar{\mathbf{X}}$ is chosen such that $\bar{\mathbf{X}}$ obeys the following rules.

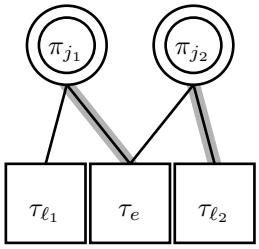
USE 1: If task $\tau_\ell \in \bar{\tau}_{\text{rdy}}(t)$ is unmatched, then each CPU $\pi_j \in \alpha_\ell$ is matched with a task τ_e with $\bar{d}_e\big(\bar{\mathbf{X}}, t\big) \leq d_\ell(t)$.

USE 2: If tasks $\tau_e, \tau_\ell \in \tau^{\text{glob}}$ are matched to CPUs π_{j_1} and π_{j_2} (*i.e.*, $\bar{x}_{e,j_1} = \bar{x}_{\ell,j_2} = 1$) and $d_e(t) < d_\ell(t)$, then $sp^{(j_1)} \geq sp^{(j_2)}$. ◁

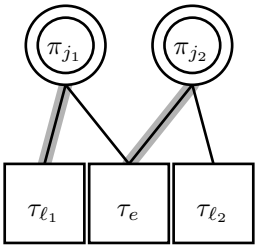
USE 1 can be thought of as an extension of the scheduling shifts in Strong-APA-EDF. To see this, consider the implications of USE 1 as $sp^{\text{L}} \to 1.0$. The platform becomes a special case of an IDENTICAL/ARBITRARY multiprocessor, for which Strong-APA-EDF is designed. We have

$$\bar{d}_i\big(\bar{\mathbf{X}}, t\big) \to \begin{cases} d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big) & \tau_i \in \tau^{\text{glob}},\ \bar{x}_{i,j} = 1,\ \pi_j \in \pi^{\text{big}},\ \text{and } d_i(t) < d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big) \\ d^{\text{big}}\big(\bar{\mathbf{X}}, t\big) & \tau_i \in \tau^{\text{glob}},\ \bar{x}_{i,j} = 1,\ \pi_j \in \pi^{\text{LIT}},\ \text{and } d_i(t) < d^{\text{big}}\big(\bar{\mathbf{X}}, t\big) \\ d_i(t) & \text{otherwise} \end{cases} .$$

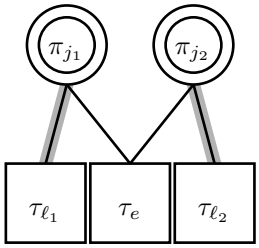
Suppose that at some time t, tasks τ_e, τ_{ℓ_1}, and τ_{ℓ_2} are such that $\tau_e \in \tau^{\text{glob}}$ is scheduled on CPU $\pi_{j_1} \in \pi^{\text{big}}$, $\tau_{\ell_1} \in \tau^{\text{big}}$ is unscheduled, and $\tau_{\ell_2} \in \tau^{\text{LIT}}$ is scheduled on CPU $\pi_{j_2} \in \pi^{\text{LIT}}$ (see Figure 5.10a). Deadlines are such that $d_e(t) < d_{\ell_1}(t) < d_{\ell_2}(t)$ and $d_{\ell_2}(t) = d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big)$, *i.e.*, task τ_{ℓ_2} has the latest deadline of any task scheduled on a CPU in π^{LIT}. Under Strong-APA-EDF, a scheduling shift would occur such that



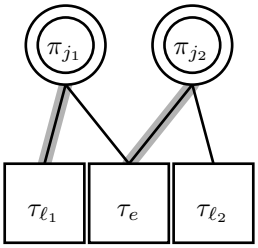
(a) Initial state.



(b) Shift under Strong-APA-EDF because $d_{\ell_1}(t) < d_{\ell_2}(t)$.



(c) τ_{ℓ_1} preempts τ_e because $d_{\ell_1}(t) < \bar{d}_e(\bar{\mathbf{X}}, t)$.



(d) τ_e preempts τ_{ℓ_2} because $d_e(t) < d_{\ell_2}(t)$.

Figure 5.10: USE 1 as an extension of Strong-APA-EDF.

π_{j1} would schedule τ_{ℓ_1} and π_{j2} would schedule τ_e (see Figure 5.10b). Under Ufm-SC-EDF, even though $d_e(t) < d_{\ell_1}(t)$, USE 1 allows τ_{ℓ_1} to preempt τ_e (see Figure 5.10c) because

$$\begin{aligned} d_{\ell_1}(t) &< d_{\ell_2}(t) \\ &= d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big) \\ &= \bar{d}_e\big(\bar{\mathbf{X}}, t\big). \end{aligned}$$

Once τ_e is preempted by τ_{ℓ_1} on π_{j_1}, task τ_e can then preempt τ_{ℓ_2} on π_{j_2} (see Figure 5.10d). Observe by comparing Figures 5.10b and 5.10d that the behavior under Ufm-SC-EDF is the same as that of Strong-APA-EDF.

USE 2 can be thought of as an extension of Ufm-EDF. If $\tau = \tau^{\text{glob}}$, then the platform becomes a special case of a UNIFORM/GLOBAL multiprocessor, for which Ufm-EDF is designed. USE 2 becomes equivalent to Ufm-EDF.

5.3.3 Ufm-SC-EDF is a Special Case of Unr-WC

Our proof that Ufm-SC-EDF is a special case of Unr-WC requires substantial setup. We recommend reviewing the discussion of symmetric differences presented in Section 3.5.1.3.

5.3.3.1 Converting Speeds between UNRELATED and UNIFORM/SEMI-CLUSTERED

To show that Ufm-SC-EDF, which targets a UNIFORM/SEMI-CLUSTERED system, is a special case of Unr-WC, which targets an UNRELATED system, we will need to convert between the notation used for the two multiprocessor models (*e.g.*, $sp^{i,j}$ versus $sp^{(j)}$). The relationships between $sp^{i,j}$, $sp^{(j)}$, 1.0 (capacity of a big CPU), and sp^{L} (capacity of a LITTLE CPU) for which of τ^{glob}, τ^{big}, or τ^{LIT} task τ_i belongs to and which of π^{big} or π^{LIT} CPU π_j belongs to are enumerated in (5.2)-(5.7) below. These equations follow from Definitions 5.1 and 5.2. Recall from the discussion in Section 1.1 that a task 'executing' on a CPU it does not have affinity for is analytically treated as executing with 0 speed.

$$\tau_i \in \tau^{\text{glob}} \wedge \pi_j \in \pi^{\text{big}} \Rightarrow sp^{i,j} = sp^{(j)} = 1.0 \tag{5.2}$$

$$\tau_i \in \tau^{\text{glob}} \wedge \pi_j \in \pi^{\text{LIT}} \Rightarrow sp^{i,j} = sp^{(j)} = sp^{\text{L}} \tag{5.3}$$

$$\tau_i \in \tau^{\text{big}} \wedge \pi_j \in \pi^{\text{big}} \Rightarrow sp^{i,j} = sp^{(j)} = 1.0 \tag{5.4}$$

$$\tau_i \in \tau^{\text{big}} \wedge \pi_j \in \pi^{\text{LIT}} \Rightarrow sp^{i,j} = 0 \tag{5.5}$$

$$\tau_i \in \tau^{\text{LIT}} \wedge \pi_j \in \pi^{\text{big}} \Rightarrow sp^{i,j} = 0 \tag{5.6}$$

$$\tau_i \in \tau^{\text{LIT}} \wedge \pi_j \in \pi^{\text{LIT}} \Rightarrow sp^{i,j} = sp^{(j)} = sp^{\text{L}} \tag{5.7}$$

5.3.3.2 Priority Points and Deadlines

To argue that we have implemented a special case of Unr-WC, we need to show that our patched `SCHED_DEADLINE` chooses configurations that optimally solve the AP instances corresponding with Unr-WC. These AP instances depend on the profit functions $\Psi_i(t)$ of each task τ_i (Definition 3.6), which themselves depend on how tasks' priority points $pp_i(t)$ are defined.

▽ **Definition 5.9.** Under Ufm-SC-EDF, the priority point of task $\tau_i \in \bar{\tau}$ is $pp_i(t) \triangleq d_i(t) - T_{[1]}$. △

Note that because each idle task $\tau_i \in \tau^{\text{idle}}$ has a well-defined deadline $d_i(t)$ (Definition 5.3), $pp_i(t)$ is also well-defined for any $\tau_i \in \tau^{\text{idle}}$.

As stated previously in this section, the choice of $pp_i(t)$ in Definition 5.9 is made over letting $pp_i(t) = d_i(t)$, as in standard EDF, to mitigate the issues discussed in Section 3.5.1.1.

▷ **Lemma 5.1.** For any time t and task $\tau_i \in \tau_{\text{rdy}}(t)$, we have $pp_i(t) \leq t$. ◁

Proof. We have

$$\left.\begin{aligned} pp_i(t) &= \{\text{Definition 5.9}\} \\ &\quad d_i(t) - T_{[1]} \\ &= a_i(t) + D_i - T_{[1]} \\ &\leq a_i(t) + D_i - T_i \\ &\leq \{\text{Constrained Deadlines Assumption}\} \\ &\quad a_i(t). \end{aligned}\right\} \tag{5.8}$$

By the No-Early-Releasing Assumption, we have $\tau_i \in \tau_{\text{rdy}}(t) \Rightarrow t \geq a_i(t)$. By (5.8), we have $\tau_i \in \tau_{\text{rdy}}(t) \Rightarrow t \geq pp_i(t)$. □

Corollaries 5.2 and 5.3 show that $t + T_{[1]}$ is an upper bound on tasks' `deadlines`, as well as $d^{\text{big}}\big(\bar{\mathbf{X}}, t\big)$ and $d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big)$. Corollary 5.2 follows from Definition 5.9 and Lemma 5.1.

▷ **Corollary 5.2.** For any time t and task $\tau_i \in \tau_{\text{rdy}}(t)$, we have $d_i(t) \leq t + T_{[1]}$. ◁

An implication of Definition 5.3 and Corollary 5.2 is that any ready real-time task in τ has no later of a deadline than any idle task in τ^{idle}.

Corollary 5.3 follows from Definitions 5.3, 5.6, and 5.7 and Corollary 5.2.

▷ **Corollary 5.3.** For any time t and configuration $\bar{\mathbf{X}}$, we have both $d^{\text{big}}\big(\bar{\mathbf{X}}, t\big) \leq t + T_{[1]}$ and $d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big) \leq t + T_{[1]}$. ◁

Lemma 5.4 relates a task's weighted deadline and deadline.

▷ **Lemma 5.4.** For any configuration $\bar{\mathbf{X}}$, time t, and task τ_i, we have $\bar{d}_i\big(\bar{\mathbf{X}}, t\big) \geq d_i(t)$. ◁

Proof. We consider three cases corresponding with the three cases in Definition 5.8.

◀ **Case 5.4.1.** $\tau_i \in \tau^{\text{glob}}$, $\bar{x}_{i,j} = 1$, $\pi_j \in \pi^{\text{big}}$, and $d_i(t) < d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big)$. ▶

We have

$$\begin{aligned}
\bar{d}_i\big(\bar{\mathbf{X}}, t\big) = \; & \{\text{Definition 5.8}\} \\
& \big(1.0 - sp^{\text{L}}\big) \cdot d_i(t) + sp^{\text{L}} \cdot d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big) \\
> \; & \big\{d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big) > d_i(t)\big\} \\
& \big(1.0 - sp^{\text{L}}\big) \cdot d_i(t) + sp^{\text{L}} \cdot d_i(t) \\
= \; & d_i(t).
\end{aligned}$$

♦

◀ **Case 5.4.2.** $\tau_i \in \tau^{\text{glob}}$, $\bar{x}_{i,j} = 1$, $\pi_j \in \pi^{\text{LIT}}$, and $d_i(t) < d^{\text{big}}\big(\bar{\mathbf{X}}, t\big)$. ▶

We have

$$\begin{aligned}
\bar{d}_i\big(\bar{\mathbf{X}}, t\big) = \; & \{\text{Definition 5.8}\} \\
& \frac{1.0}{sp^{\text{L}}} \cdot d^{\text{big}}\big(\bar{\mathbf{X}}, t\big) - \frac{1.0 - sp^{\text{L}}}{sp^{\text{L}}} \cdot d_i(t) \\
> \; & \big\{d^{\text{big}}\big(\bar{\mathbf{X}}, t\big) > d_i(t)\big\} \\
& \frac{1.0}{sp^{\text{L}}} \cdot d_i(t) - \frac{1.0 - sp^{\text{L}}}{sp^{\text{L}}} \cdot d_i(t) \\
= \; & \frac{sp^{\text{L}}}{sp^{\text{L}}} \cdot d_i(t) \\
= \; & d_i(t).
\end{aligned}$$

♦

◀ **Case 5.4.3.** Neither of the conditions in Cases 5.4.1 and 5.4.2 are true. ▶

By Definition 5.8, we have $\bar{d}_i\big(\bar{\mathbf{X}}, t\big) = d_i(t)$. ♦

In all cases, we have $\bar{d}_i\big(\bar{\mathbf{X}}, t\big) \geq d_i(t)$. □

5.3.3.3 Profit

We now present how *profit* is defined for idle tasks. Using this definition, we will augment the profit matrix of the AP instance that corresponds with Unr-WC to yield an instance that considers all tasks in $\bar{\tau}$. We

do this because it will be easier to prove that Ufm-SC-EDF yields configurations that optimally solve this augmented AP instance.

▽ **Definition 5.10.** The *profit* of idle task $\tau_i \in \tau^{\text{idle}}$ is $\Psi_i(t) \triangleq 0$. △

▷ **Lemma 5.5.** Consider any time t. Let $\text{AP}(\tau, \pi, \mathbf{P})$ denote the AP instance corresponding with Unr-WC at time t. Consider the optimization problem $\text{AP}\big(\bar{\tau}, \pi, \bar{\mathbf{P}}\big)$ such that

$$\bar{\mathbf{P}} \triangleq \begin{bmatrix} \Psi_1(t) \cdot sp^{1,1} & \Psi_1(t) \cdot sp^{1,2} & \dots & \Psi_1(t) \cdot sp^{1,m} \\ \Psi_2(t) \cdot sp^{2,1} & \Psi_2(t) \cdot sp^{2,2} & & \\ \vdots & & \ddots & \\ \Psi_{n+m}(t) \cdot sp^{n+m,1} & & & \Psi_{n+m}(t) \cdot sp^{n+m,m} \end{bmatrix}.$$

$\text{AP}\big(\bar{\tau}, \pi, \bar{\mathbf{P}}\big)$ differs from $\text{AP}(\tau, \pi, \mathbf{P})$ in that $\text{AP}\big(\bar{\tau}, \pi, \bar{\mathbf{P}}\big)$ considers idle tasks. Let $\mathbf{X}$ and $\bar{\mathbf{X}}$ be a configuration and its corresponding augmented configuration. $\mathbf{X}$ is an optimal solution of $\text{AP}(\tau, \pi, \mathbf{P})$ if and only if $\bar{\mathbf{X}}$ is an optimal solution of $\text{AP}\big(\bar{\tau}, \pi, \bar{\mathbf{P}}\big)$. ◁

Proof. By (2.1), the objective function value of a solution of $\text{AP}\big(\bar{\tau}, \pi, \bar{\mathbf{P}}\big)$ is

$$\begin{aligned} \sum_{\tau_i \in \bar{\tau}} \sum_{\pi_j \in \pi} \Psi_i(t) \cdot sp^{i,j} \cdot \bar{x}_{i,j} &= \big\{\text{By Definition 5.4, } \bar{\tau} = \tau \cup \tau^{\text{idle}}\big\} \\ &\quad \left(\sum_{\tau_i \in \tau} \sum_{\pi_j \in \pi} \Psi_i(t) \cdot sp^{i,j} \cdot \bar{x}_{i,j} \right) + \left(\sum_{\tau_i \in \tau^{\text{idle}}} \sum_{\pi_j \in \pi} \Psi_i(t) \cdot sp^{i,j} \cdot \bar{x}_{i,j} \right) \\ &= \{\text{Definition 5.10}\} \\ &\quad \left(\sum_{\tau_i \in \tau} \sum_{\pi_j \in \pi} \Psi_i(t) \cdot sp^{i,j} \cdot \bar{x}_{i,j} \right) + \left(\sum_{\tau_i \in \tau^{\text{idle}}} \sum_{\pi_j \in \pi} 0 \cdot sp^{i,j} \cdot \bar{x}_{i,j} \right) \\ &= \sum_{\tau_i \in \tau} \sum_{\pi_j \in \pi} \Psi_i(t) \cdot sp^{i,j} \cdot \bar{x}_{i,j} \\ &= \big\{\text{By Definition 5.5, for any } \tau_i \in \tau \text{ and } \pi_j \in \pi, \bar{x}_{i,j} = x_{i,j}\big\} \\ &\quad \sum_{\tau_i \in \tau} \sum_{\pi_j \in \pi} \Psi_i(t) \cdot sp^{i,j} \cdot x_{i,j}. \end{aligned}$$

Thus, the objective function values of $\mathbf{X}$ for $\text{AP}(\tau, \pi, \mathbf{P})$ and $\bar{\mathbf{X}}$ for $\text{AP}\big(\bar{\tau}, \pi, \bar{\mathbf{P}}\big)$ are equal. The lemma follows. □

▷ **Lemma 5.6.** For any task $\tau_i \in \bar{\tau}$, we have

$$\Psi_i(t) = \begin{cases} t + T_{[1]} - d_i(t) & \tau_i \in \bar{\tau}_{\text{rdy}}(t) \\ 0 & \tau_i \notin \bar{\tau}_{\text{rdy}}(t) \end{cases}. \qquad \triangleleft$$

Proof. If $\tau_i \in \tau^{\text{idle}}$, we have

$$\begin{aligned} \Psi_i(t) = & \{\text{Definition 5.10}\} \\ & 0 \\ = & t + T_{[1]} - t - T_{[1]} \\ = & \{\text{Definition 5.3}\} \\ & t + T_{[1]} - d_i(t) \\ = & \left\{\tau_i \in \tau^{\text{idle}} \text{ and, by Definition 5.4, } \tau^{\text{idle}} \subseteq \bar{\tau}_{\text{rdy}}(t)\right\} \\ & \begin{cases} t + T_{[1]} - d_i(t) & \tau_i \in \bar{\tau}_{\text{rdy}}(t) \\ 0 & \tau_i \notin \tau_{\text{rdy}}(t) \end{cases}. \end{aligned}$$

The remaining possibility is $\tau_i \notin \tau^{\text{idle}}$. Here, we have $\tau_i \in \tau$. Because $\tau_i \notin \tau^{\text{idle}}$, by Definition 5.4, we have

$$\tau_i \in \bar{\tau}_{\text{rdy}}(t) \Leftrightarrow \tau_i \in \tau_{\text{rdy}}(t). \tag{5.9}$$

We have

$$\begin{aligned} \Psi_i(t) = & \{\text{Definition 3.6}\} \\ & \begin{cases} t - pp_i(t) & t > pp_i(t) \text{ and } \tau_i \in \tau_{\text{rdy}}(t) \\ 0 & t \leq pp_i(t) \text{ or } \tau_i \notin \tau_{\text{rdy}}(t) \end{cases} \\ = & \{\text{If } t = pp_i(t), \text{ then } t - pp_i(t) = 0\} \end{aligned}$$

$$
\begin{cases} t - pp_i(t) & t \geq pp_i(t) \text{ and } \tau_i \in \tau_{\text{rdy}}(t) \\ 0 & t < pp_i(t) \text{ or } \tau_i \notin \tau_{\text{rdy}}(t) \end{cases}
$$

$= \{\text{Lemma 5.1}\}$

$$
\begin{cases} t - pp_i(t) & \tau_i \in \tau_{\text{rdy}}(t) \\ 0 & t < pp_i(t) \text{ or } \tau_i \notin \tau_{\text{rdy}}(t) \end{cases}
$$

$= \left\{\tau_i \notin \tau_{\text{rdy}}(t) \Rightarrow \left(t < pp_i(t) \text{ or } \tau_i \notin \tau_{\text{rdy}}(t)\right)\right\}$

$$
\begin{cases} t - pp_i(t) & \tau_i \in \tau_{\text{rdy}}(t) \\ 0 & \tau_i \notin \tau_{\text{rdy}}(t) \end{cases}
$$

$= \{\text{Definition 5.9}\}$

$$
\begin{cases} t + T_{[1]} - d_i(t) & \tau_i \in \tau_{\text{rdy}}(t) \\ 0 & \tau_i \notin \tau_{\text{rdy}}(t) \end{cases}
$$

$= \{\text{Equation (5.9)}\}$

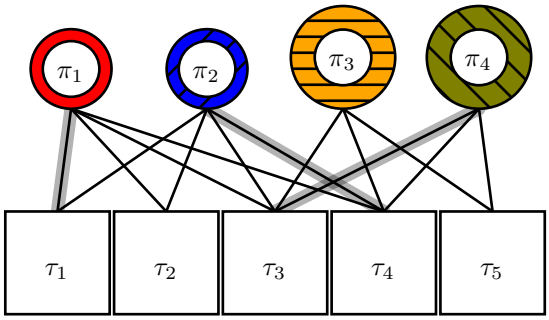
$$
\begin{cases} t + T_{[1]} - d_i(t) & \tau_i \in \bar{\tau}_{\text{rdy}}(t) \\ 0 & \tau_i \notin \bar{\tau}_{\text{rdy}}(t) \end{cases}.
$$

The lemma is true whether $\tau_i \in \tau^{\text{idle}}$ or $\tau_i \in \tau$. This proves the lemma. □

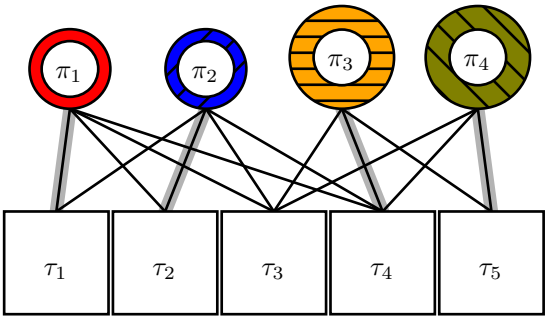
5.3.3.4 Connected Components

Our proof that Ufm-SC-EDF is a special case of Unr-WC will involve reasoning about the symmetric difference between the configurations chosen by Ufm-SC-EDF and Unr-WC. Rather than considering the entire symmetric difference, it will be more convenient to reason about a particular connected component of the symmetric difference. We introduce new notation for connected components.

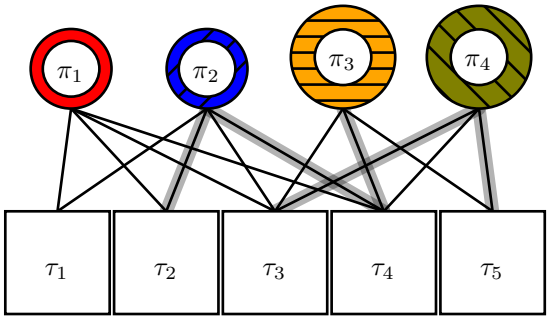
▼ **Example 5.6.** Consider the matchings $\mathbb{M}$, illustrated in Figure 5.11a, and $\mathbb{M}'$, illustrated in Figure 5.11b. The symmetric difference (recall Definition 3.7 in Section 3.5.1.3) $\mathbb{M}\Delta\mathbb{M}'$ is illustrated in Figure 5.11c. This symmetric difference contains two connected components: the paths $(\tau_2, \pi_2, \tau_4, \pi_3)$ and (τ_3, π_4, τ_5).



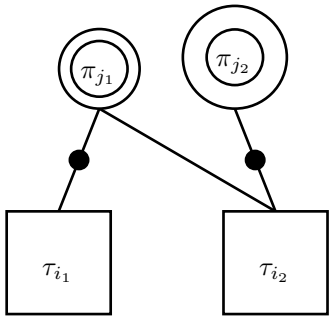
(a) Matching $\mathbb{M}$.



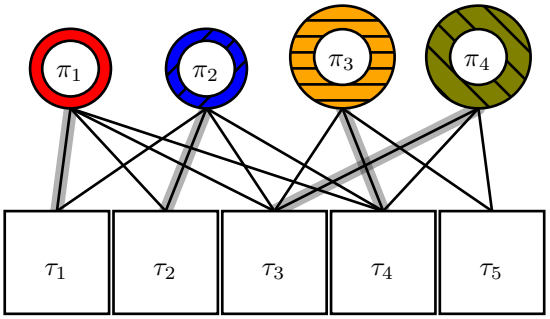
(b) Matching $\mathbb{M}'$.



(c) Symmetric difference $\mathbb{M}\Delta\mathbb{M}'$.



(d) Structure of connected component $(\tau_2, \pi_2, \tau_4, \pi_3)$. Dots (•) denote edges in $\mathbb{M}'$.



(e) $\mathbb{M}$ with component $(\tau_2, \pi_2, \tau_4, \pi_3)$ inverted.

Figure 5.11: Illustration of a connected component.

Figure 5.11d shows the structure of connected component $(\tau_2, \pi_2, \tau_4, \pi_3)$ with $i_1 \leftarrow 2$, $i_2 \leftarrow 4$, $j_1 \leftarrow 2$, and $j_2 \leftarrow 3$. In Figure 5.11d and later figures that illustrate components, task and CPU indices are written indirectly (*i.e.*, as i_1, i_2, ... instead of 1, 2, ...) because, in later proofs (Lemmas 5.21 and 5.22) that reason about the structure of connected components, the specific tasks and CPUs in said components are unknown. A dot ($\bullet$) is used to differentiate which edges in the component belong to which matching (*e.g.*, dots denote edges from $\mathbb{M}'$ in Figure 5.11d). ▲

A connected component in a symmetric difference is assigned a numerical value called its contribution.

▽ **Definition 5.11.** Let $\mathbf{X}$ and $\mathbf{X}'$ be two solutions to an AP instance with profit matrix $\mathbf{P}$. Let $\mathbb{M}$ and $\mathbb{M}'$ be the matchings corresponding with $\bar{\mathbf{X}}$ and $\bar{\mathbf{X}}'$, respectively. Let edge set $\mathbb{E} \subseteq \mathbb{M}\Delta\mathbb{M}'$ be the edges of a path or cycle. The *contribution* of $\mathbb{E}$ from $\mathbb{M}$ to $\mathbb{M}'$ is

$$\sum_{(\tau_i,\pi_j)\in\mathbb{E}} \left(\begin{cases} p_{i,j} & (\tau_i, \pi_j) \in \mathbb{M}' \\ -p_{i,j} & (\tau_i, \pi_j) \in \mathbb{M} \end{cases} \right).$$

The contribution of a connected component in $\mathbb{M}\Delta\mathbb{M}'$ from $\mathbb{M}$ to $\mathbb{M}'$ is the contribution of the edges in this connected component. △

Note that the contribution of an edge set $\mathbb{E}$ from $\mathbb{M}'$ to $\mathbb{M}$ is the negative of its contribution from $\mathbb{M}$ to $\mathbb{M}'$.

The contribution of $\mathbb{E}$ from $\mathbb{M}$ to $\mathbb{M}'$ is equivalent to the change in objective function value of $\bar{\mathbf{X}}$ caused by inverting each edge in $\mathbb{E}$ (*i.e.*, for each edge $(\tau_i, \pi_j) \in \mathbb{E}$, setting $\bar{x}_{i,j} \leftarrow 1$ if $(\tau_i, \pi_j) \in \mathbb{M}'$ and $\bar{x}_{i,j} \leftarrow 0$ if $(\tau_i, \pi_j) \in \mathbb{M}$). This is demonstrated in the following example.

▼ **Example 5.6 (continued).** As in the system illustrated in Figure 5.11, let $\tau = \{\tau_1, \tau_2, \ldots, \tau_5\}$ and $\pi = \{\pi_1, \pi_2, \ldots, \pi_4\}$. Consider optimization problem AP$(\tau, \pi, \mathbf{P})$, where

$$\mathbf{P} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 2 & 2 & 0 & 0 \\ 3 & 3 & 6 & 6 \\ 4 & 4 & 8 & 8 \\ 0 & 0 & 10 & 10 \end{bmatrix}.$$

The contribution of component $(\tau_2, \pi_2, \tau_4, \pi_3)$ (Figure 5.11d) in symmetric difference $\mathbb{M}\Delta\mathbb{M}'$ is (recall that the edges in $\mathbb{M}'$ are denoted by dots in Figure 5.11d) $p_{2,2} + p_{4,3} - p_{4,2} = 2 + 8 - 4 = 6$.

Observe that matching $\mathbb{M}$ has objective function value $p_{1,1} + p_{3,4} + p_{4,2} = 1 + 6 + 4 = 11$. The matching yielded by inverting component $(\tau_2, \pi_2, \tau_4, \pi_3)$ in matching $\mathbb{M}$, which is shown in Figure 5.11e, has value $p_{1,1} + p_{2,2} + p_{3,4} + p_{4,3} = 1 + 2 + 6 + 8 = 17$. The increase in objective function value is $17 - 11 = 6$, which matches the contribution of $(\tau_2, \pi_2, \tau_4, \pi_3)$ computed earlier. ▲

5.3.3.5 Relabeling

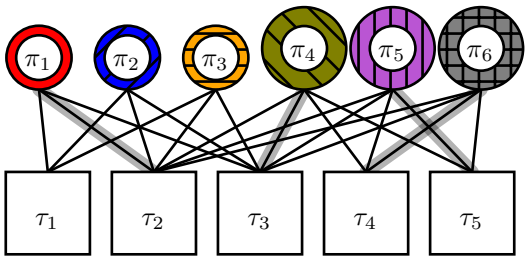
Later on we will reason about the symmetric difference between configurations selected by Ufm-SC-EDF and Unr-WC. We introduce a procedure we call relabeling (Algorithm 2) that simplifies this symmetric difference. Given two configurations $\mathbf{X}$ and $\mathbf{X}'$, the goal of relabeling $\mathbf{X}'$ by $\mathbf{X}$ is to swap task-to-CPU matchings in $\mathbf{X}'$ to more closely resemble those in $\mathbf{X}$.

For simplicity, relabeling is a procedure that takes and returns non-augmented configurations. Relabeling between augmented configurations is implicitly defined because there is a one-to-one mapping between configurations and augmented configurations. For example, suppose we have configurations $\bar{\mathbf{X}}^{(1)}$ and $\bar{\mathbf{X}}^{(2)}$. These have non-augmented configurations $\mathbf{X}^{(1)}$ and $\mathbf{X}^{(2)}$. Let $\mathbf{X}^{(3)}$, with corresponding augmented configuration $\bar{\mathbf{X}}^{(3)}$, denote the relabeling of $\mathbf{X}^{(2)}$ by $\mathbf{X}^{(1)}$. $\bar{\mathbf{X}}^{(3)}$ is the relabeling of $\bar{\mathbf{X}}^{(2)}$ by $\bar{\mathbf{X}}^{(1)}$.

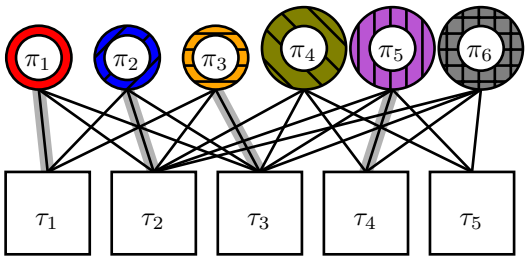
As we present the following definitions, lemmas, and corollaries (*i.e.*, those in Section 5.3.3.5) that consider relabeling between configurations, note that they all concern only the tasks in τ. Recall that, by Definition 5.5, any task in τ that is matched to a CPU in a configuration $\mathbf{X}$ is also matched to this same CPU in augmented configuration $\bar{\mathbf{X}}$. Thus, these definitions, lemmas, and corollaries also apply to relabeling between augmented configurations.

▼ **Example 5.7.** Consider a system with five tasks, three LITTLE CPUs, and three big CPUs. Consider the configurations $\mathbf{X}$, illustrated in Figure 5.12a, and $\mathbf{X}'$, illustrated in Figure 5.12b. Figure 5.12 illustrates the *relabeling* (Algorithm 2) of $\mathbf{X}'$ via $\mathbf{X}$. Initially, output configuration $\mathbf{X}^*$ is set to $\mathbf{X}'$.

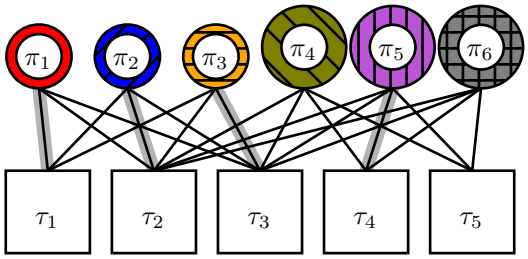
Consider the first iteration ($i \leftarrow 1$) of the **for** loop at line 3. Task τ_1 is unmatched in configuration $\mathbf{X}$. Thus, the condition of the **if** statement at line 4 is false. Configuration $\mathbf{X}^*$ is unchanged, as illustrated in Figures 5.12b and 5.12c.



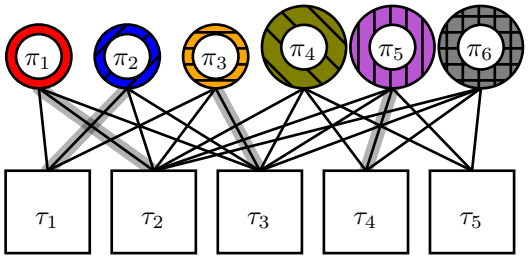
(a) Configuration $\mathbf{X}$.



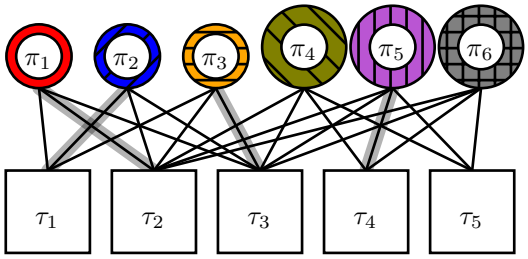
(b) Configuration $\mathbf{X}'$.



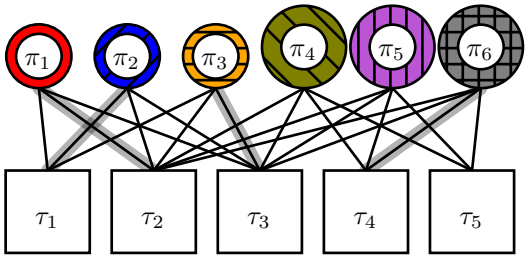
(c) Configuration $\mathbf{X}^*$ (first iteration).



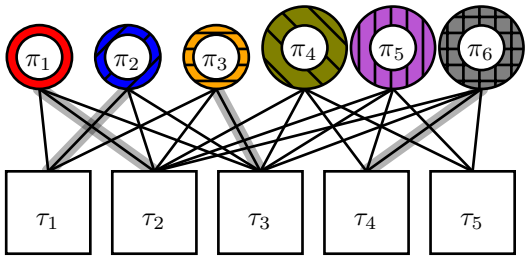
(d) Configuration $\mathbf{X}^*$ (second iteration).



(e) Configuration $\mathbf{X}^*$ (third iteration).



(f) Configuration $\mathbf{X}^*$ (fourth iteration).



(g) Configuration $\mathbf{X}^*$ (fifth iteration).

Figure 5.12: Relabeling example.

```
1  Function Relabel(X', X):
2      X* ← X'
3      for i ← 1 ... n do
4          if ∃π_{j_i}, π_{j_i^*} ∈ π : x_{i,j_i} = 1 and x*_{i,j_i^*} = 1 then        // τ_i matched in X and X*
5              if π_{j_i}, π_{j_i^*} ∈ π^big or π_{j_i}, π_{j_i^*} ∈ π^LIT then
6                  if ∃τ_{i^*} ∈ τ : x*_{i^*,j_i} = 1 then
7                      x*_{i^*,j_i} ← 0                    // τ_{i^*} unmatched from π_{j_i}
8                      x*_{i^*,j_i^*} ← 1                  // τ_{i^*} matched with π_{j_i^*}
9                  end if
10                 x*_{i,j_i^*} ← 0                        // τ_i unmatched from π_{j_i^*}
11                 x*_{i,j_i} ← 1                          // τ_i matched with π_{j_i}
12             end if
13         end if
14     end for
15     return X*
16 end
```

Algorithm 2: Configuration relabeling.

Consider the second iteration ($i \leftarrow 2$). Task τ_2 is matched with CPU π_1 in configuration $\mathbf{X}$ ($j_2 \leftarrow 1$) and on CPU π_2 in configuration $\mathbf{X}^*$ ($j_2^* \leftarrow 2$). The condition on line 4 is satisfied. The **if** block at line 5 is entered because CPUs π_1 and π_2 are both LITTLE CPUs. Because CPU π_1 is already scheduling task τ_1 in configuration $\mathbf{X}^*$ ($i^* \leftarrow 1$), the **if** block at line 6 is entered. Task τ_1 is moved to CPU π_2 (lines 7 and 8). Task τ_2 is moved to CPU π_1 (lines 10 and 11). Configuration $\mathbf{X}^*$ is as illustrated in Figure 5.12d.

Consider the third iteration ($i \leftarrow 3$). Task τ_3 is matched with CPU π_4 in configuration $\mathbf{X}$ ($j_3 \leftarrow 4$) and on CPU π_3 in configuration $\mathbf{X}^*$ ($j_3^* \leftarrow 3$). The condition on line 4 is satisfied. The condition on line 5 is false because CPU π_3 is LITTLE and CPU π_4 is big. Configuration $\mathbf{X}^*$ is unchanged in this iteration.

Consider the fourth iteration ($i \leftarrow 4$). Task τ_4 is matched with CPU π_6 in configuration $\mathbf{X}$ ($j_4 \leftarrow 6$) and on CPU π_5 in configuration $\mathbf{X}^*$ ($j_4^* \leftarrow 5$). The condition on line 4 is satisfied. The condition on line 5 is satisfied because both CPUs π_5 and π_6 are big. CPU π_6 is free in configuration $\mathbf{X}^*$, so task τ_4 is moved to CPU π_6. Configuration $\mathbf{X}^*$ is as illustrated in Figure 5.12f.

In the fifth iteration ($i \leftarrow 5$), task τ_5 is unmatched in configuration $\mathbf{X}^*$. Thus, configuration $\mathbf{X}^*$ is unchanged in this iteration. The **for** loop at line 3, as well as function `Relabel`, terminate. ▲

▷ **Lemma 5.7.** Consider configurations $\mathbf{X}$ and $\mathbf{X}'$. After any iteration of the **for** loop at line 3 in `Relabel`, any task τ_i is matched with CPU π_j in $\mathbf{X}'$ (*i.e.*, $x'_{i,j} = 1$) if and only if τ_i is matched with a CPU π_{j^*} in $\mathbf{X}^*$ (*i.e.*, $x^*_{i,j^*} = 1$) such that $\pi_j, \pi_{j^*} \in \pi^{\text{big}}$ or $\pi_j, \pi_{j^*} \in \pi^{\text{LIT}}$. ◁

Proof. We prove by induction. For the base case, consider the state of `Relabel` immediately after line 2, *i.e.*, after zero iterations of the **for** loop at line 3. Because $\mathbf{X}^*$ is set to $\mathbf{X}'$, the lemma statement is true after zero iterations.

Consider the i^{th} iteration of the **for** loop for some $i > 0$. Assume that the lemma statement has not been violated by the beginning of the i^{th} iteration. In this iteration, tasks τ_i and τ_{i^*} (if it exists) swap their matched CPUs π_{j_i} and $\pi_{j_i^*}$ only if the condition on line 5 is true. Thus, the swap only occurs if π_{j_i} and $\pi_{j_i^*}$ are such that $\pi_{j_i}, \pi_{j_i^*} \in \pi^{\text{big}}$ or $\pi_{j_i}, \pi_{j_i^*} \in \pi^{\text{LIT}}$. Because the lemma statement was not violated prior to the i^{th} iteration, the lemma statement cannot be violated due to this swap. Because this swap is the only change in $\mathbf{X}^*$ made in the i^{th} iteration, the i^{th} iteration does not violate the lemma statement. This completes the proof by induction. □

▷ **Lemma 5.8.** Consider configurations $\mathbf{X}$ and $\mathbf{X}'$. Let $\mathbf{X}^* = \texttt{Relabel}(\mathbf{X}', \mathbf{X})$. For any time t, $\mathbf{X}'$ and $\mathbf{X}^*$ have equal objective function value for the AP instance corresponding with Unr-WC at time t. ◁

Proof. $\mathbf{X}^*$ is initialized to $\mathbf{X}'$. Thus, $\mathbf{X}'$ and $\mathbf{X}^*$ initially have equal objective function value. Consider the change in value of $\mathbf{X}^*$ due to a single iteration of the **for** loop at line 3.

Line 7 decreases the value by $sp^{(j_i)} \cdot \Psi_{i^*}(t)$ and line 8 increases the value by $sp^{(j_i^*)} \cdot \Psi_{i^*}(t)$. By Lemma 5.7 and Definition 5.2, we have $sp^{(j_i)} = sp^{(j_i^*)}$. Thus, the net change by lines 7 and 8 is 0.

Line 10 decreases the value by $sp^{(j_i^*)} \cdot \Psi_i(t)$ and line 11 increases the value by $sp^{(j_i)} \cdot \Psi_i(t)$. By Lemma 5.7 and Definition 5.2, we have $sp^{(j_i^*)} = sp^{(j_i)}$. Thus, the net change by lines 10 and 11 is 0.

The net change in any iteration of the **for** loop is 0. Thus, the net change by `Relabel` is 0. Because $\mathbf{X}'$ and $\mathbf{X}^*$ began with equal objective function value, they yield equal value after `Relabel` completes. □

▽ **Definition 5.12.** The *similarity* between configurations $\mathbf{X}$ and $\mathbf{X}'$ is the number of tasks $\tau_i \in \tau$ such that τ_i is matched with big CPUs in both $\mathbf{X}$ and $\mathbf{X}'$ or with LITTLE CPUs in both $\mathbf{X}$ and $\mathbf{X}'$. △

Note that idle tasks in τ^{idle} do not count towards similarity between configurations. Thus, the similarity between $\mathbf{X}$ and $\mathbf{X}'$ is equivalent to the similarity between augmented configurations $\bar{\mathbf{X}}$ and $\bar{\mathbf{X}}'$.

Corollary 5.9 follows from Lemma 5.7 and Definition 5.12.

▷ **Corollary 5.9.** Consider configurations $\mathbf{X}$ and $\mathbf{X}'$. Let $\mathbf{X}^* = \texttt{Relabel}(\mathbf{X}', \mathbf{X})$. The similarity between $\mathbf{X}$ and $\mathbf{X}'$ is equal to the similarity between $\mathbf{X}$ and $\mathbf{X}^*$. ◁

▷ **Lemma 5.10.** Consider configurations $\mathbf{X}$ and $\mathbf{X}'$. Let $\mathbf{X}^* = \texttt{Relabel}(\mathbf{X}', \mathbf{X})$. If task $\tau_i \in \tau$ is matched with π_j in configuration $\mathbf{X}$ (*i.e.*, $x_{i,j} = 1$) and on $\pi_{j'}$ in configuration $\mathbf{X}'$ (*i.e.*, $x'_{i,j'} = 1$) such that $\pi_j, \pi_{j'} \in \pi^{\text{big}}$ or $\pi_j, \pi_{j'} \in \pi^{\text{LIT}}$, then task τ_i is matched with π_j in $\mathbf{X}^*$ (*i.e.*, $x^*_{i,j} = 1$). ◁

Proof. Let task τ_{i_1} be such that τ_{i_1} is matched with π_j in $\mathbf{X}$ and with $\pi_{j'}$ in $\mathbf{X}'$, *i.e.*, $x_{i_1,j} = x'_{i_1,j'} = 1$, and either $\pi_j, \pi_{j'} \in \pi^{\text{big}}$ or $\pi_j, \pi_{j'} \in \pi^{\text{LIT}}$. Our proof obligation is to show that task τ_{i_1} is matched with CPU π_j in $\mathbf{X}^*$, *i.e.*, $x^*_{i_1,j} = 1$.

We prove by induction that $x^*_{i_1,j} = 1$ after the i^{th} iteration of the **for** loop at line 3 for any i such that $i_1 \leq i \leq n$.

For the base case, consider the $(i_1)^{\text{th}}$ iteration ($i \leftarrow i_1$). Because $x'_{i_1,j'} = 1$, by Lemma 5.7, at the beginning of the $(i_1)^{\text{th}}$ iteration, task τ_{i_1} is matched with some CPU π_{j^*} in $\mathbf{X}^*$ (*i.e.*, $x^*_{i_1,j^*} = 1$) such that either $\pi_{j'}, \pi_{j^*} \in \pi^{\text{big}}$ or $\pi_{j'}, \pi_{j^*} \in \pi^{\text{LIT}}$. Because $x^*_{i_1,j^*} = 1$ and we have assumed that task τ_{i_1} is such that $x_{i_1,j} = 1$, the condition on line 4 is true with $j_i \leftarrow j$ and $j^*_i \leftarrow j^*$. Because task τ_{i_1} is such that $\pi_j, \pi_{j'} \in \pi^{\text{big}}$ or $\pi_j, \pi_{j'} \in \pi^{\text{LIT}}$, we have that $\pi_j, \pi_{j^*} \in \pi^{\text{big}}$ or $\pi_j, \pi_{j^*} \in \pi^{\text{LIT}}$. Thus, the condition on line 5 is also satisfied.

Because the conditions on lines 4 and 5 are satisfied, lines 10-11 are executed in this iteration. Because $i \leftarrow i_1$ and $j_i \leftarrow j$ in this iteration, the execution of line 11 sets $x^*_{i_1,j} = 1$. This is the proof obligation of the base case.

For the induction step, consider the $(i_2)^{\text{th}}$ iteration ($i \leftarrow i_2$) such that $i_1 < i_2 \leq n$ and $x^*_{i_1,j} = 1$ at the beginning of this iteration. It remains to show that task τ_{i_1} is never unmatched from π_j in $\mathbf{X}^*$ in this iteration. Task τ_{i_1} can only be unmatched from CPU π_j during lines 7-8 because lines 10-11 only affect task τ_{i_2}. For lines 7-8 to have been executed in this iteration and affect the matching of task τ_{i_1} in $\mathbf{X}^*$, the condition on line 6 must be true with $i^* \leftarrow i_1$.

We prove that this condition is not true with $i^* \leftarrow i_1$ by contradiction. Assume otherwise that $x^*_{i_1,j_i} = 1$. We have assumed, by induction, that $x^*_{i_1,j} = 1$. Because $x^*_{i_1,j} = 1$, $x^*_{i_1,j_i} = 1$, and each task is matched to at most one CPU in $\mathbf{X}^*$, we must have that $j_i = j$. For lines 7-8 to have been executed, the condition on line 4 must have been true. Thus, $1 = x_{i,j_i} = x_{i_2,j}$. Because we have assumed that task τ_{i_1}

is such that $x_{i_1,j} = 1$ and each CPU is matched with at most one task in $\mathbf{X}$, we must have $i_2 = i_1$. This contradicts that $i_2 > i_1$.

Thus, the condition on line 6 is not true with $i^* \leftarrow i_1$. Lines 7-8 cannot affect the matching of task τ_{i_1} in $\mathbf{X}^*$. No lines in the $(i_2)^{\text{th}}$ iteration can affect the matching of task τ_{i_1} in $\mathbf{X}^*$. Thus, $x^*_{i_1,j} = 1$ remains true after the $(i_2)^{\text{th}}$ iteration. This completes the proof by induction. □

5.3.3.6 Proving Ufm-SC-EDF is a Special Case of Unr-WC

We prove that Ufm-SC-EDF is a special case of Unr-WC by examining the structure of the symmetric difference between configurations selected by the two schedulers. We will show that each connected component in this symmetric difference has one of four possible structures. By reasoning about the contribution of these connected components, we will prove that at least one of USE 1 or USE 2 must have been broken if the configuration selected by Unr-WC has a higher objective function value than the configuration selected by Ufm-SC-EDF.

▽ **Definition 5.13.** Let $\bar{\mathbf{X}}^{\text{USE}}(t)$ denote the augmented configuration selected by Ufm-SC-EDF at time t.

Let $\bar{\mathbf{X}}^{\text{opt}}(t)$ denote the augmented configuration such that $\mathbf{X}^{\text{opt}}(t) = \texttt{Relabel}(\mathbf{X}, \mathbf{X}^{\text{USE}}(t))$ where $\mathbf{X}$ is an optimal configuration for the AP instance corresponding with Unr-WC at time t that, among all optimal configurations, has maximum similarity to $\mathbf{X}^{\text{USE}}(t)$.

Let $\bar{\mathbb{M}}^{\text{USE}}(t)$ and $\bar{\mathbb{M}}^{\text{opt}}(t)$ denote the matchings that correspond with $\bar{\mathbf{X}}^{\text{USE}}(t)$ and $\bar{\mathbf{X}}^{\text{opt}}(t)$, respectively. △

▷ **Corollary 5.11.** $\bar{\mathbf{X}}^{\text{opt}}(t)$ is an optimal solution of $\text{AP}\big(\bar{\tau}, \pi, \bar{\mathbf{P}}\big)$, where profit matrix $\bar{\mathbf{P}}$ is as defined in Lemma 5.6. ◁

Proof. The corollary follows from Definition 5.13 and Lemmas 5.5 and 5.8. □

▷ **Corollary 5.12.** $\bar{\mathbf{X}}^{\text{opt}}(t)$ is the optimal solution of $\text{AP}\big(\bar{\tau}, \pi, \bar{\mathbf{P}}\big)$, where $\bar{\mathbf{P}}$ is as defined in Lemma 5.6, that has maximum similarity to $\bar{\mathbf{X}}^{\text{USE}}(t)$. ◁

Proof. The corollary follows from Definition 5.13 and Corollaries 5.9 and 5.11. □

▷ **Corollary 5.13.** If task $\tau_i \in \bar{\tau}$ is matched on CPU π_{j_1} in $\bar{\mathbb{M}}^{\text{USE}}(t)$ and on CPU π_{j_2} in $\bar{\mathbb{M}}^{\text{opt}}(t)$ such that $\pi_{j_1}, \pi_{j_2} \in \pi^{\text{big}}$ or $\pi_{j_1}, \pi_{j_2} \in \pi^{\text{LIT}}$, then $\pi_{j_1} = \pi_{j_2}$. ◁

Proof. Recall that $\bar{\tau} = \tau \cup \tau^{\text{idle}}$. The corollary is true for any task $\tau_i \in \tau$ by Lemma 5.10. For any $\tau_i \in \tau^{\text{idle}}$, by Definition 5.3, task τ_i has affinity for only one CPU π_j. If matched in both $\bar{\mathbb{M}}^{\text{USE}}(t)$ and $\bar{\mathbb{M}}^{\text{opt}}(t)$, task τ_i must be matched to π_j in both $\bar{\mathbb{M}}^{\text{USE}}(t)$ and $\bar{\mathbb{M}}^{\text{opt}}(t)$. □

▷ **Lemma 5.14.** Every task τ_i in a non-trivial (*i.e.*, contains more than a single node) path in symmetric difference $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$ is such that $\tau_i \in \bar{\tau}_{\text{rdy}}(t)$. ◁

Proof. If $\tau_i \in \tau^{\text{idle}}$, then by Definition 5.4, we have $\tau_i \in \bar{\tau}_{\text{rdy}}(t)$.

If task $\tau_i \in \tau$ is in a non-trivial path, it is incident on at least one edge in $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$. By Definition 3.7, task τ_i must be matched in at least one of $\bar{\mathbb{M}}^{\text{USE}}(t)$ and $\bar{\mathbb{M}}^{\text{opt}}(t)$. Because $\bar{\mathbb{M}}^{\text{USE}}(t)$ and $\bar{\mathbb{M}}^{\text{opt}}(t)$ correspond with (canonical) configurations $\bar{\mathbf{X}}^{\text{USE}}(t)$ and $\bar{\mathbf{X}}^{\text{opt}}(t)$, by Definition 2.26, we have $\tau_i \in \tau_{\text{rdy}}(t)$. By Definition 5.4, we have $\tau_i \in \bar{\tau}_{\text{rdy}}(t)$. □

▷ **Lemma 5.15.** Every CPU π_j is incident on either zero or two edges in $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$. ◁

Proof. $\bar{\mathbf{X}}^{\text{USE}}(t)$ and $\bar{\mathbf{X}}^{\text{opt}}(t)$ are both augmented configurations. By Definition 5.5, in both matchings $\bar{\mathbb{M}}^{\text{USE}}(t)$ and $\bar{\mathbb{M}}^{\text{opt}}(t)$, π_j is matched with either a task $\tau_{\text{rdy}}(t)$ or a task in τ^{idle}. Regardless of whether such tasks are in $\tau_{\text{rdy}}(t)$ or τ^{idle}, CPU π_j is matched with some task τ_{i_1} in $\bar{\mathbb{M}}^{\text{USE}}(t)$ and task τ_{i_2} in $\bar{\mathbb{M}}^{\text{opt}}(t)$. Stated formally, $(\tau_{i_1}, \pi_j) \in \bar{\mathbb{M}}^{\text{USE}}(t)$ and $(\tau_{i_2}, \pi_j) \in \bar{\mathbb{M}}^{\text{opt}}(t)$.

If $\tau_{i_1} = \tau_{i_2}$, then $(\tau_{i_1}, \pi_j) \in \bar{\mathbb{M}}^{\text{USE}}(t) \cap \bar{\mathbb{M}}^{\text{opt}}(t)$. By Definition 3.7, we have $(\tau_{i_1}, \pi_j) \notin \bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$. Thus, π_j is incident on zero edges in $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$.

Otherwise, we have $\tau_{i_1} \neq \tau_{i_2}$. Then, by Definition 3.7, we have both $(\tau_{i_1}, \pi_j) \in \bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$ and $(\tau_{i_2}, \pi_j) \in \bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$. Thus, π_j is incident on two edges in $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$. □

▷ **Lemma 5.16.** Every task $\tau_i \in \tau^{\text{idle}}$ is incident on at most one edge in $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$. ◁

Proof. By Definition 5.3, each task $\tau_i \in \tau^{\text{idle}}$ has affinity for one CPU. Let this CPU be π_j. Because $\bar{\mathbf{X}}^{\text{USE}}(t)$ and $\bar{\mathbf{X}}^{\text{opt}}(t)$ are configurations, they are canonical at t (see Definition 2.26), and thus can only be matched with π_j in both $\bar{\mathbb{M}}^{\text{USE}}(t)$ and $\bar{\mathbb{M}}^{\text{opt}}(t)$. Thus, τ_i can only be incident on (τ_i, π_j) in $\bar{\mathbb{M}}^{\text{USE}}(t) \cup \bar{\mathbb{M}}^{\text{opt}}(t)$. By Definition 3.7, $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$ is a subset of $\bar{\mathbb{M}}^{\text{USE}}(t) \cup \bar{\mathbb{M}}^{\text{opt}}(t)$, thus τ_i can only be incident on (τ_i, π_j) in $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$. □

▷ **Lemma 5.17.** Every task $\tau_i \in \tau^{\text{big}} \cup \tau^{\text{LIT}}$ is incident on at most one edge in $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$. ◁

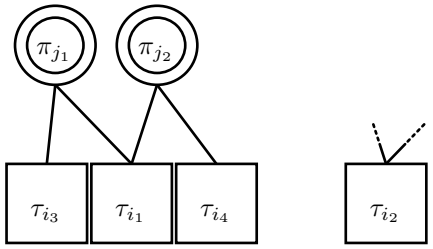


Figure 5.13: Tasks τ_{i_1} and τ_{i_2} exist in distinct connected components.

Proof. Task τ_i is incident on at most two edges in $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$ because, by Definition 3.7, $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t) \subseteq \bar{\mathbb{M}}^{\text{USE}}(t) \cup \bar{\mathbb{M}}^{\text{opt}}(t)$ and each task is matched at most once in any matching. We prove that τ_i is not incident on two edges by contradiction. Task τ_i is incident on two edges in $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$ if τ_i is matched to different CPUs in $\bar{\mathbb{M}}^{\text{USE}}(t)$ and $\bar{\mathbb{M}}^{\text{opt}}(t)$. Suppose task τ_i is matched to π_{j_1} in $\bar{\mathbb{M}}^{\text{USE}}(t)$ and to π_{j_2} in $\bar{\mathbb{M}}^{\text{opt}}(t)$ such that $\pi_{j_1} \neq \pi_{j_2}$. Because $\tau_i \in \tau^{\text{big}} \cup \tau^{\text{LIT}}$, τ_i has affinity for either only CPUs in π^{big} or only CPUs in π^{LIT}. Thus, either $\pi_{j_1}, \pi_{j_2} \in \pi^{\text{big}}$ or $\pi_{j_1}, \pi_{j_2} \in \pi^{\text{LIT}}$. That $\pi_{j_1} \neq \pi_{j_2}$ contradicts Corollary 5.13. □

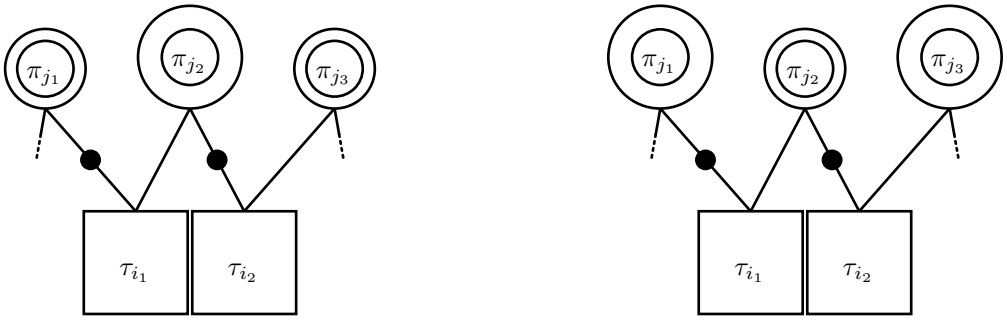
▷ **Lemma 5.18.** If task $\tau_i \in \tau^{\text{glob}}$ is incident on two edges in $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$, then these edges are (τ_i, π_{j_1}) and (τ_i, π_{j_2}) such that $\pi_{j_1} \in \pi^{\text{big}}$ and $\pi_{j_2} \in \pi^{\text{LIT}}$. ◁

Proof. Suppose otherwise that $\pi_{j_1}, \pi_{j_2} \in \pi^{\text{big}}$ or $\pi_{j_1}, \pi_{j_2} \in \pi^{\text{LIT}}$. Assume without loss of generality that task τ_i is matched with π_{j_1} in $\bar{\mathbf{X}}^{\text{USE}}(t)$ and with π_{j_2} in $\bar{\mathbf{X}}^{\text{opt}}(t)$. By Corollary 5.13, we have $\pi_{j_1} = \pi_{j_2}$. Thus, $(\tau_i, \pi_{j_1}) = (\tau_i, \pi_{j_2})$, *i.e.*, the 'edges' are actually the same edge. This contradicts that τ_i is incident on two edges in $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$. □

▷ **Lemma 5.19.** In each connected component of $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$, at most one task $\tau_i \in \tau^{\text{glob}}$ is incident on two edges in $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$. ◁

Proof. We prove by contradiction. Suppose otherwise that there are multiple tasks in τ^{glob} that are incident on two edges in $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$ in the same connected component.

► **Claim 5.19.1.** There exist two tasks $\tau_{i_1}, \tau_{i_2} \in \tau^{\text{glob}}$ in the same connected component that are both incident on two edges and have edges to the same CPU π_j. ◄



(a) $\pi_{j_2} \in \pi^{\text{big}}$.

(b) $\pi_{j_2} \in \pi^{\text{LIT}}$.

Figure 5.14: Component with two tasks in τ^{glob} that are incident on two edges. Dots (•) denote edges in $\bar{\mathbb{M}}^{\text{opt}}(t)$.

Proof. Suppose otherwise that for any such tasks τ_{i_1} and τ_{i_2}, τ_{i_1} and τ_{i_2} do not have edges to the same CPU. We will prove that tasks τ_{i_1} and τ_{i_2} exist in distinct connected components (with structure as illustrated in Figure 5.13), which contradicts the claim.

By the definition of task τ_{i_1}, task τ_{i_1} has edges to two distinct CPUs π_{j_1} and π_{j_2}. By Lemma 5.15, both π_{j_1} and π_{j_2} are incident on two edges. For both π_{j_1} and π_{j_2}, one of these edges is known to be to task τ_{i_1}. Let τ_{i_3} be the other task connected to π_{j_1} and τ_{i_4} be the other task connected to π_{j_2}.

It must be the case that τ_{i_3} is incident on at most one edge. If $\tau_{i_3} \in \tau^{\text{glob}}$, then by the supposition made at the beginning of this claim proof, task τ_{i_3} is incident on at most one edge. Otherwise, $\tau_{i_3} \notin \tau^{\text{glob}}$, which implies $\tau_{i_3} \in \tau^{\text{idle}} \cup \tau^{\text{big}} \cup \tau^{\text{LIT}}$. By Lemmas 5.16 and 5.17, τ_{i_3} is , again, incident on at most one edge.

It is already known that τ_{i_3} has an edge to π_{j_1}, so (τ_{i_3}, π_{j_1}) must be the only edge τ_{i_3} is incident on. By the same reasoning for τ_{i_4} and π_{j_2}, it must be that (τ_{i_4}, π_{j_2}) is the only edge τ_{i_4} is incident on.

Thus, $(\tau_{i_3}, \pi_{j_1}, \tau_{i_1}, \pi_{j_2}, \tau_{i_4})$ is the entirety of the connected component containing τ_{i_1} (see Figure 5.13). Task τ_{i_1} is unconnected with task τ_{i_2}. This contradicts that τ_{i_1} and τ_{i_2} are in the same connected component. ■

As illustrated in the insets of Figure 5.14, let tasks τ_{i_1} and τ_{i_2} be as defined in Claim 5.19.1 such that π_{j_2} is their shared CPU. Let π_{j_1} be the other CPU sharing an edge with τ_{i_1} and π_{j_3} be the other CPU

sharing an edge with τ_{i_2}. These tasks and CPUs form a path $(\pi_{j_1}, \tau_{i_1}, \pi_{j_2}, \tau_{i_2}, \pi_{j_3})$.[3] By Lemma 5.18, if $\pi_{j_2} \in \pi^{\text{big}}$, then $\pi_{j_1}, \pi_{j_3} \in \pi^{\text{LIT}}$, and, if $\pi_{j_2} \in \pi^{\text{LIT}}$, then $\pi_{j_1}, \pi_{j_3} \in \pi^{\text{big}}$. The structure of path $(\pi_{j_1}, \tau_{i_1}, \pi_{j_2}, \tau_{i_2}, \pi_{j_3})$ must match one of the insets of Figure 5.14. Note that we can assume without loss of generality that $\{(\tau_{i_1}, \pi_{j_1}), (\tau_{i_2}, \pi_{i_2})\} \subseteq \bar{\mathbb{M}}^{\text{opt}}(t)$ and $\{(\tau_{i_1}, \pi_{j_2}), (\tau_{i_2}, \pi_{j_3})\} \subseteq \bar{\mathbb{M}}^{\text{USE}}(t)$. This is because if this assumption is invalid, then we can swap the indexes of tasks τ_{i_1} and τ_{i_2} to make it valid.

We consider four cases depending on the sign of the contribution of $(\pi_{j_1}, \tau_{i_1}, \pi_{j_2}, \tau_{i_2}, \pi_{j_3})$ and whether $\pi_{j_2} \in \pi^{\text{big}}$ or $\pi_{j_2} \in \pi^{\text{LIT}}$.

◀ **Case 5.19.1.** For $\text{AP}\big(\bar{\tau}, \pi, \bar{\mathbf{P}}\big)$, the contribution of path $(\pi_{j_1}, \tau_{i_1}, \pi_{j_2}, \tau_{i_2}, \pi_{j_3})$ from $\bar{\mathbb{M}}^{\text{USE}}(t)$ to $\bar{\mathbb{M}}^{\text{opt}}(t)$ is negative. ▶

By Definition 5.11, the contribution of $(\pi_{j_1}, \tau_{i_1}, \pi_{j_2}, \tau_{i_2}, \pi_{j_3})$ from $\bar{\mathbb{M}}^{\text{opt}}(t)$ to $\bar{\mathbb{M}}^{\text{USE}}(t)$ is positive. This means that inverting the edges of $(\pi_{j_1}, \tau_{i_1}, \pi_{j_2}, \tau_{i_2}, \pi_{j_3})$ in $\bar{\mathbb{M}}^{\text{opt}}(t)$ results in a matching that corresponds to a solution of $\text{AP}\big(\bar{\tau}, \pi, \bar{\mathbf{P}}\big)$ with a greater objective function value than $\bar{\mathbf{X}}^{\text{opt}}(t)$. This contradicts Corollary 5.11. ♦

◀ **Case 5.19.2.** For $\text{AP}\big(\bar{\tau}, \pi, \bar{\mathbf{P}}\big)$, the contribution of path $(\pi_{j_1}, \tau_{i_1}, \pi_{j_2}, \tau_{i_2}, \pi_{j_3})$ from $\bar{\mathbb{M}}^{\text{USE}}(t)$ to $\bar{\mathbb{M}}^{\text{opt}}(t)$ is zero. ▶

By Definition 5.11, the contribution of $(\pi_{j_1}, \tau_{i_1}, \pi_{j_2}, \tau_{i_2}, \pi_{j_3})$ from $\bar{\mathbb{M}}^{\text{opt}}(t)$ to $\bar{\mathbb{M}}^{\text{USE}}(t)$ is zero. This means that inverting the edges of $(\pi_{j_1}, \tau_{i_1}, \pi_{j_2}, \tau_{i_2}, \pi_{j_3})$ in $\bar{\mathbb{M}}^{\text{opt}}(t)$ results in a matching that corresponds to a solution of $\text{AP}\big(\bar{\tau}, \pi, \bar{\mathbf{P}}\big)$ with an equal objective function value to that of $\bar{\mathbf{X}}^{\text{opt}}(t)$. Inverting $(\pi_{j_1}, \tau_{i_1}, \pi_{j_2}, \tau_{i_2}, \pi_{j_3})$ in $\bar{\mathbb{M}}^{\text{opt}}(t)$ also matches task τ_{i_1} to CPU π_{j_2} and task τ_{i_2} to CPU π_{j_3}, as in $\bar{\mathbb{M}}^{\text{USE}}(t)$ (in Figure 5.14, inverting this path effectively replaces the edges (τ_{i_1}, π_{j_1}) and (τ_{i_2}, π_{j_2}) in $\bar{\mathbb{M}}^{\text{opt}}(t)$ with the edges (τ_{i_1}, π_{j_2}) and (τ_{i_2}, π_{j_3}) in $\bar{\mathbb{M}}^{\text{USE}}(t)$). By Definition 5.12, inverting path $(\pi_{j_1}, \tau_{i_1}, \pi_{j_2}, \tau_{i_2}, \pi_{j_3})$ in $\bar{\mathbb{M}}^{\text{opt}}(t)$ increases the similarity of the configuration with $\bar{\mathbf{X}}^{\text{USE}}(t)$ by two. This contradicts Corollary 5.12. ♦

◀ **Case 5.19.3.** For $\text{AP}\big(\bar{\tau}, \pi, \bar{\mathbf{P}}\big)$, the contribution of path $(\pi_{j_1}, \tau_{i_1}, \pi_{j_2}, \tau_{i_2}, \pi_{j_3})$ from $\bar{\mathbb{M}}^{\text{USE}}(t)$ to $\bar{\mathbb{M}}^{\text{opt}}(t)$ is positive and $\pi_{j_2} \in \pi^{\text{big}}$. ▶

[3]Note that, though we state that $(\pi_{j_1}, \tau_{i_1}, \pi_{j_2}, \tau_{i_2}, \pi_{j_3})$ is a path, we have not excluded the possibility that $\pi_{j_1} = \pi_{j_3}$, which would make this path a cycle. The remaining logic of the proof of this lemma remains unchanged even if this path is actually a cycle.

Note that, because $\pi_{j_2} \in \pi^{\text{big}}$, the structure of $(\pi_{j_1}, \tau_{i_1}, \pi_{j_2}, \tau_{i_2}, \pi_{j_3})$ is as in Figure 5.14a.

By Definition 5.11, we have

$$\left.\begin{aligned} 0 &< \bar{p}_{i_1,j_1} - \bar{p}_{i_1,j_2} + \bar{p}_{i_2,j_2} - \bar{p}_{i_2,j_3} \\ &= \Psi_{i_1}(t) \cdot sp^{i_1,j_1} - \Psi_{i_1}(t) \cdot sp^{i_1,j_2} + \Psi_{i_2}(t) \cdot sp^{i_2,j_2} - \Psi_{i_2}(t) \cdot sp^{i_2,j_3}. \end{aligned}\right\} \tag{5.10}$$

Because $\tau_{i_1}, \tau_{i_2} \in \tau^{\text{glob}}$, by (5.2) and (5.3), we have $sp^{i_1,j_1} = sp^{(j_1)}$, $sp^{i_1,j_2} = sp^{(j_2)}$, $sp^{i_2,j_2} = sp^{(j_2)}$, and $sp^{i_2,j_3} = sp^{(j_3)}$.

We have

$$\left.\begin{aligned} 0 <\ & \{\text{Equation (5.10)}\} \\ & \Psi_{i_1}(t) \cdot sp^{i_1,j_1} - \Psi_{i_1}(t) \cdot sp^{i_1,j_2} + \Psi_{i_2}(t) \cdot sp^{i_2,j_2} - \Psi_{i_2}(t) \cdot sp^{i_2,j_3} \\ =\ & \Psi_{i_1}(t) \cdot sp^{(j_1)} - \Psi_{i_1}(t) \cdot sp^{(j_2)} + \Psi_{i_2}(t) \cdot sp^{(j_2)} - \Psi_{i_2}(t) \cdot sp^{(j_3)} \\ =\ & \Psi_{i_1}(t) \cdot \left(sp^{(j_1)} - sp^{(j_2)}\right) - \Psi_{i_2}(t) \cdot \left(sp^{(j_3)} - sp^{(j_2)}\right) \\ =\ & \left\{ sp^{(j_1)} = sp^{(j_3)} \text{ (see Figure 5.14)} \right\} \\ & \Psi_{i_1}(t) \cdot \left(sp^{(j_1)} - sp^{(j_2)}\right) - \Psi_{i_2}(t) \cdot \left(sp^{(j_1)} - sp^{(j_2)}\right). \end{aligned}\right\} \tag{5.11}$$

Note that the above derivations of (5.10) and (5.11) will also hold in Case 5.19.4.

Because $\pi_{j_2} \in \pi^{\text{big}}$, we have $sp^{(j_1)} - sp^{(j_2)} = sp^{\text{L}} - 1.0 < 0$. By (5.11), we have $\Psi_{i_1}(t) < \Psi_{i_2}(t)$. By Lemmas 5.6 and 5.14, we have $d_{i_1}(t) > d_{i_2}(t)$.

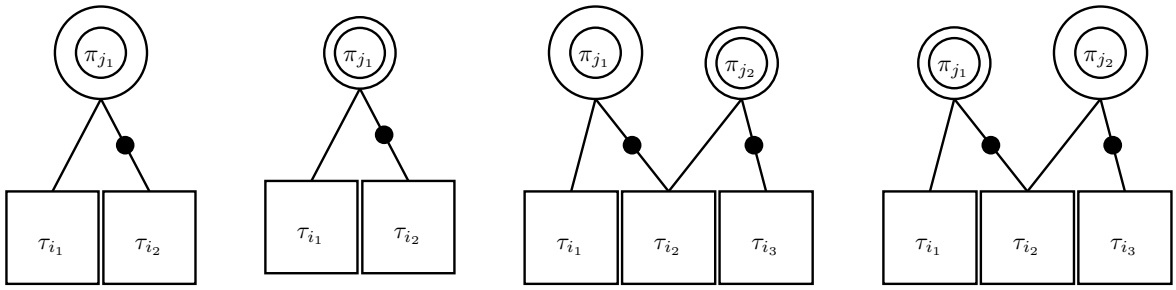
Recall from Figure 5.14a that, in $\bar{\mathbb{M}}^{\text{USE}}(t)$, τ_{i_1} is matched with $\pi_{j_2} \in \pi^{\text{big}}$ and τ_{i_2} is matched with $\pi_{j_3} \in \pi^{\text{LIT}}$. Because $\tau_{i_1}, \tau_{i_2} \in \tau^{\text{glob}}$ and $d_{i_1}(t) > d_{i_2}(t)$, this violates USE 2. This contradicts the definition of $\bar{\mathbf{X}}^{\text{USE}}(t)$ (Definition 5.13). ♦

◄ **Case 5.19.4.** For $\text{AP}\big(\bar{\tau}, \pi, \bar{\mathbf{P}}\big)$, the contribution of path $(\pi_{j_1}, \tau_{i_1}, \pi_{j_2}, \tau_{i_2}, \pi_{j_3})$ from $\bar{\mathbb{M}}^{\text{USE}}(t)$ to $\bar{\mathbb{M}}^{\text{opt}}(t)$ is positive and $\pi_{j_2} \in \pi^{\text{LIT}}$. ►

Note that, because $\pi_{j_2} \in \pi^{\text{LIT}}$, the structure of $(\pi_{j_1}, \tau_{i_1}, \pi_{j_2}, \tau_{i_2}, \pi_{j_3})$ is as in Figure 5.14b.

Because, as in Case 5.19.3, the contribution of $(\pi_{j_1}, \tau_{i_1}, \pi_{j_2}, \tau_{i_2}, \pi_{j_3})$ is positive, we have that (5.11) is true.

Because $\pi_{j_2} \in \pi^{\text{LIT}}$, we have $sp^{(j_1)} - sp^{(j_2)} = 1.0 - sp^{\text{L}} > 0$. By (5.11), we have $\Psi_{i_1}(t) > \Psi_{i_2}(t)$. By Lemmas 5.6 and 5.14, we have $d_{i_1}(t) < d_{i_2}(t)$.



(a) (b) (c) (d)

Figure 5.15: All possible cases for connected components in $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$. Edges marked with • denote edges in $\bar{\mathbb{M}}^{\text{USE}}(t)$, while unmarked edges denote edges in $\bar{\mathbb{M}}^{\text{opt}}(t)$.

Recall from Figure 5.14b that, in $\bar{\mathbb{M}}^{\text{USE}}(t)$, τ_{i_1} is matched with $\pi_{j_2} \in \pi^{\text{LIT}}$ and τ_{i_2} is matched with $\pi_{j_3} \in \pi^{\text{big}}$. Because $\tau_{i_1}, \tau_{i_2} \in \tau^{\text{glob}}$ and $d_{i_1}(t) < d_{i_2}(t)$, this violates USE 2. This contradicts the definition of $\bar{\mathbf{X}}^{\text{USE}}(t)$ (Definition 5.13). ♦

All cases yield a contradiction, completing the proof of the lemma. □

▷ **Lemma 5.20.** All non-trivial connected components in $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$ are paths. ◁

Proof. By Lemma 3.27, every connected component is either a path, unconnected node, or cycle. It is sufficient to prove that no connected component is a cycle. In an undirected bipartite graph, a cycle must include at least two task nodes. Each task in the cycle must be incident on two edges. By Lemmas 5.16, 5.17, and 5.19, at most one task in any connected component is incident on two edges in $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$. Every node in a cycle must be incident on two edges. A non-trivial cycle in an undirected bipartite graph must contain, at minimum, two tasks. Thus, no connected component in $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$ is a cycle. □

▷ **Lemma 5.21.** Every non-trivial connected component in $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$ has structure as illustrated in one of the insets of Figure 5.15. ◁

Proof. Consider an arbitrary non-trivial connected component in $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$. By Lemma 5.20, this connected component is a path. We first prove that this path has structure as illustrated in one of the insets of Figure 5.15. We will initially ignore which edges in the component belong to $\bar{\mathbb{M}}^{\text{USE}}(t)$ or $\bar{\mathbb{M}}^{\text{opt}}(t)$ (*i.e.*, which edges are highlighted with dots in the insets of Figure 5.15).

By Lemma 5.15, any CPU π_j has two edges in the path (if π_j has zero edges, it cannot be in the path). Thus, any ending node of the path, which must be incident on only a single edge, must be a task. Let task τ_{i_1} be an ending node of the path. Because τ_{i_1} is incident on an edge, it must be connected to some CPU by this edge. Let this CPU be π_{j_1}. By Lemma 5.15, π_{j_1} has another edge, and is thus connected to another task besides τ_{i_1}. Let this task be τ_{i_2}.

Because τ_{i_2} is a node in a path, it is incident on either one or two edges. There are four cases depending on whether $\pi_{j_1} \in \pi^{\text{big}}$ or $\pi_{j_1} \in \pi^{\text{LIT}}$ and whether τ_{i_2} has one or two edges. Each case corresponds with an inset of Figure 5.15.

◀ **Case 5.21.1.** $\pi_{j_1} \in \pi^{\text{big}}$ and τ_{i_2} is incident on one edge. ▶

The structure of the path is as illustrated in Figure 5.15a. ♦

◀ **Case 5.21.2.** $\pi_{j_1} \in \pi^{\text{LIT}}$ and τ_{i_2} is incident on one edge. ▶

The structure of the path is as illustrated in Figure 5.15b. ♦

◀ **Case 5.21.3.** $\pi_{j_1} \in \pi^{\text{big}}$ and τ_{i_2} is incident on two edges. ▶

Because task τ_{i_2} is incident on two edges, by Lemmas 5.16 and 5.17, we have $\tau_{i_2} \notin \tau^{\text{idle}} \cup \tau^{\text{big}} \cup \tau^{\text{LIT}}$. The only remaining possibility is that $\tau_{i_2} \in \tau^{\text{glob}}$. Let π_{j_2} denote the CPU incident on the edge of τ_{i_2} that is not (τ_{i_2}, π_{j_1}). Because $\pi_{j_1} \in \pi^{\text{big}}$, by Lemma 5.18, we have $\pi_{j_2} \in \pi^{\text{LIT}}$. By Lemma 5.15, π_{j_2} is connected to another task by an edge. Let this task be τ_{i_3}. By Lemmas 5.16, 5.17, and 5.19, we have that τ_{i_3} is incident on at most one edge. Thus, τ_{i_3} must be an endpoint of the path. The structure of the path in this case is as illustrated in Figure 5.15c. ♦

◀ **Case 5.21.4.** $\pi_{j_1} \in \pi^{\text{LIT}}$ and τ_{i_2} is incident on two edges. ▶

The reasoning of this case closely resembles that of Case 5.21.3. Because task τ_{i_2} is incident on two edges, we have $\tau_{i_2} \in \tau^{\text{glob}}$. Let π_{j_2} denote the other CPU sharing an edge with τ_{i_2}. Because $\pi_{j_1} \in \pi^{\text{LIT}}$, by Lemma 5.18, we have $\pi_{j_2} \in \pi^{\text{big}}$. By Lemma 5.15, π_{j_2} is connected to another task by an edge. Let this task be τ_{i_3}. By Lemmas 5.16, 5.17, and 5.19, we have that τ_{i_3} is incident on at most one edge. Thus, τ_{i_3} must be an endpoint of the path. The structure of the path in this case is as illustrated in Figure 5.15c. ♦

It remains to prove that the positions of dots in Figure 5.15 are sufficient to cover all possible cases of connected components. Recall that each edge in a connected component in $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$ belongs to either $\bar{\mathbb{M}}^{\text{USE}}(t)$ or $\bar{\mathbb{M}}^{\text{opt}}(t)$. Also recall that any node is matched at most once in a matching. Thus, each matching contributes at most one edge to every node in the path. This means the path is alternating for both $\bar{\mathbb{M}}^{\text{USE}}(t)$ and $\bar{\mathbb{M}}^{\text{opt}}(t)$. Dot placements on edges must either be as in the insets in Figure 5.15 or the opposite (*e.g.*, in Figure 5.15a, removing the dot on (τ_{i_2}, π_{j_1}) and placing a dot on (τ_{i_1}, π_{j_1}), and in Figure 5.15c, removing the dots on (τ_{i_2}, π_{j_1}) and (τ_{i_3}, π_{j_2}) and adding dots on (τ_{i_1}, π_{j_1}) and (τ_{i_2}, π_{j_2})). Note that the opposite dot placement of Figure 5.15a is a mirror image of Figure 5.15a. Thus, Figure 5.15a and its opposite dot placement are isomorphic (*i.e.*, the graphs are identical under re-indexing). Likewise, the opposite dot placement of Figure 5.15b is also isomorphic with Figure 5.15b. The opposite dot placement of Figure 5.15c is isomorphic with Figure 5.15d, and vice versa. Because each possible alternative dot placement is isomorphic to an inset, the insets of Figure 5.15 are sufficient to account for all possible dot placements. □

▷ **Lemma 5.22.** Ufm-SC-EDF is a special case of Unr-WC. ◁

Proof. We prove by contradiction. Suppose otherwise that Ufm-SC-EDF is not a special case of Unr-WC. Then for some time instant t, configuration $\mathbf{X}^{\text{USE}}(t)$ is not an optimal solution of the AP instance corresponding with Unr-WC at time t. By Lemma 5.5, $\bar{\mathbf{X}}^{\text{USE}}(t)$ is not an optimal solution of $\text{AP}\big(\bar{\tau}, \pi, \bar{\mathbf{P}}\big)$. By Corollary 5.11, $\bar{\mathbf{X}}^{\text{opt}}(t)$ has a higher objective function value than $\bar{\mathbf{X}}^{\text{USE}}(t)$. Thus, there must exist a connected component in $\bar{\mathbb{M}}^{\text{USE}}(t)\Delta\bar{\mathbb{M}}^{\text{opt}}(t)$ with positive contribution from $\bar{\mathbb{M}}^{\text{USE}}(t)$ to $\bar{\mathbb{M}}^{\text{opt}}(t)$.

◀ **Case 5.22.1.** The positive connected component has structure as in Figures 5.15a or 5.15b. ▶

Because the connected component is positive, we must have $\Psi_{i_1}(t) \cdot sp^{(j_1)} > \Psi_{i_2}(t) \cdot sp^{(j_1)}$. By dividing both sides by $sp^{(j_1)}$, we have $\Psi_{i_1}(t) > \Psi_{i_2}(t)$. By Lemmas 5.6 and 5.14, we have $t + T_{[1]} - d_{i_1}(t) > t + T_{[1]} - d_{i_2}(t)$. This implies that $d_{i_1}(t) < d_{i_2}(t)$. By Lemma 5.4, we have $d_{i_1}(t) < \bar{d}_{i_2}\big(\bar{\mathbf{X}}, t\big)$. This contradicts USE 1. ♦

◀ **Case 5.22.2.** The positive connected component has structure as in Figure 5.15c. ▶

Because the connected component is positive, we must have

$$\Psi_{i_1}(t) \cdot 1.0 + \Psi_{i_2}(t) \cdot sp^{\mathrm{L}} > \Psi_{i_2}(t) \cdot 1.0 + \Psi_{i_3}(t) \cdot sp^{\mathrm{L}}.$$

Simplification and rearrangement yields

$$\Psi_{i_1}(t) > \Psi_{i_2}(t) \cdot \big(1.0 - sp^{\mathrm{L}}\big) + \Psi_{i_3}(t) \cdot sp^{\mathrm{L}}.$$

By Lemmas 5.6 and 5.14, we have

$$t + T_{[1]} - d_{i_1}(t) > \big(t + T_{[1]} - d_{i_2}(t)\big) \cdot \big(1.0 - sp^{\mathrm{L}}\big) + \big(t + T_{[1]} - d_{i_3}(t)\big) \cdot sp^{\mathrm{L}}.$$

Simplification and rearrangement yields

$$d_{i_1}(t) < d_{i_2}(t) \cdot \big(1.0 - sp^{\mathrm{L}}\big) + d_{i_3}(t) \cdot sp^{\mathrm{L}}.$$

Because task τ_{i_3} is matched to CPU $\pi_{j_2} \in \pi^{\mathrm{LIT}}$ in $\bar{\mathbf{X}}^{\mathrm{USE}}(t)$, by Definitions 5.6 and 5.7, we have $d^{\mathrm{LIT}}\big(\bar{\mathbf{X}}^{\mathrm{USE}}(t), t\big) \geq d_{i_3}(t)$. Thus,

$$d_{i_1}(t) < d_{i_2}(t) \cdot \big(1.0 - sp^{\mathrm{L}}\big) + d^{\mathrm{LIT}}\big(\bar{\mathbf{X}}^{\mathrm{USE}}(t), t\big) \cdot sp^{\mathrm{L}}. \tag{5.12}$$

We will show that, by (5.12), regardless of whether $d_{i_2}(t) < d^{\mathrm{LIT}}\big(\bar{\mathbf{X}}^{\mathrm{USE}}(t), t\big)$ or $d_{i_2}(t) \geq d^{\mathrm{LIT}}\big(\bar{\mathbf{X}}^{\mathrm{USE}}(t), t\big)$, we have $d_{i_1}(t) < \bar{d}_{i_2}\big(\bar{\mathbf{X}}^{\mathrm{USE}}(t), t\big)$. Suppose we have $d_{i_2}(t) < d^{\mathrm{LIT}}\big(\bar{\mathbf{X}}^{\mathrm{USE}}(t), t\big)$. Recall that we have assumed the connected component of interest is as illustrated in Figure 5.15c. Because τ_{i_2} is incident on two edges in Figure 5.15c, by Lemmas 5.16 and 5.17, we have $\tau_{i_2} \in \tau^{\mathrm{glob}}$. Because τ_{i_2} is matched to big CPU π_{j_1} in $\bar{\mathbb{M}}^{\mathrm{USE}}(t)$ in Figure 5.15c, we have $\bar{x}^{\mathrm{USE}}_{i_2,j_1}(t) = 1$ and $\pi_{j_1} \in \pi^{\mathrm{big}}$. Thus,

$$\begin{aligned}
d_{i_1}(t) < {} & \{\text{Equation (5.12)}\} \\
& d_{i_2}(t) \cdot \big(1.0 - sp^{\mathrm{L}}\big) + d^{\mathrm{LIT}}\big(\bar{\mathbf{X}}^{\mathrm{USE}}(t), t\big) \cdot sp^{\mathrm{L}} \\
= {} & \big\{\text{Definition 5.8, } d_{i_2}(t) < d^{\mathrm{LIT}}\big(\bar{\mathbf{X}}^{\mathrm{USE}}(t), t\big), \tau_{i_2} \in \tau^{\mathrm{glob}}, \bar{x}^{\mathrm{USE}}_{i_2,j_1}(t) = 1, \text{ and } \pi_{j_1} \in \pi^{\mathrm{big}}\big\}
\end{aligned}$$

$\bar{d}_{i_2}\big(\bar{\mathbf{X}}^{\text{USE}}(t), t\big)$.

Suppose we instead have $d_{i_2}(t) \geq d^{\text{LIT}}\big(\bar{\mathbf{X}}^{\text{USE}}(t), t\big)$. We have

$$\begin{aligned}
d_{i_1}(t) < & \ \{\text{Equation (5.12)}\} \\
& d_{i_2}(t) \cdot \big(1.0 - sp^{\text{L}}\big) + d^{\text{LIT}}\big(\bar{\mathbf{X}}^{\text{USE}}(t), t\big) \cdot sp^{\text{L}} \\
\leq & \ d_{i_2}(t) \cdot \big(1.0 - sp^{\text{L}}\big) + d_{i_2}(t) \cdot sp^{\text{L}} \\
= & \ d_{i_2}(t) \\
\leq & \ \{\text{Lemma 5.4}\} \\
& \bar{d}_{i_2}\big(\bar{\mathbf{X}}^{\text{USE}}(t), t\big).
\end{aligned}$$

Regardless of whether $d_{i_2}(t) < d^{\text{LIT}}\big(\bar{\mathbf{X}}^{\text{USE}}(t), t\big)$ or $d_{i_2}(t) \geq d^{\text{LIT}}\big(\bar{\mathbf{X}}^{\text{USE}}(t), t\big)$, we have $d_{i_1}(t) < \bar{d}_{i_2}\big(\bar{\mathbf{X}}^{\text{USE}}(t), t\big)$. This contradicts USE 1. ♦

◄ **Case 5.22.3.** The positive connected component has structure as in Figure 5.15d. ►

Because the connected component is positive, we must have

$$\Psi_{i_1}(t) \cdot sp^{\text{L}} + \Psi_{i_2}(t) \cdot 1.0 > \Psi_{i_2}(t) \cdot sp^{\text{L}} + \Psi_{i_3}(t) \cdot 1.0.$$

Simplification and rearrangement yields

$$\Psi_{i_1}(t) \cdot sp^{\text{L}} > \Psi_{i_3}(t) - \Psi_{i_2}(t) \cdot \big(1.0 - sp^{\text{L}}\big).$$

Dividing both sides by sp^{L} yields

$$\Psi_{i_1}(t) > \Psi_{i_3}(t) \cdot \frac{1.0}{sp^{\text{L}}} - \Psi_{i_2}(t) \cdot \frac{1.0 - sp^{\text{L}}}{sp^{\text{L}}}.$$

By Lemmas 5.6 and 5.14, we have

$$t + T_{[1]} - d_{i_1}(t) > \big(t + T_{[1]} - d_{i_3}(t)\big) \cdot \frac{1.0}{sp^{\text{L}}} - \big(t + T_{[1]} - d_{i_2}(t)\big) \cdot \frac{1.0 - sp^{\text{L}}}{sp^{\text{L}}}.$$

Simplification and rearrangement yields

$$d_{i_1}(t) < d_{i_3}(t) \cdot \frac{1.0}{sp^{\mathrm{L}}} - d_{i_2}(t) \cdot \frac{1.0 - sp^{\mathrm{L}}}{sp^{\mathrm{L}}}.$$

Because task τ_{i_3} is matched to CPU $\pi_{j_2} \in \pi^{\text{big}}$ in $\bar{\mathbf{X}}^{\text{USE}}(t)$, by Definitions 5.6 and 5.7, we have $d^{\text{big}}\big(\bar{\mathbf{X}}^{\text{USE}}(t), t\big) \geq d_{i_3}(t)$. Thus,

$$d_{i_1}(t) < d^{\text{big}}\big(\bar{\mathbf{X}}^{\text{USE}}(t), t\big) \cdot \frac{1.0}{sp^{\mathrm{L}}} - d_{i_2}(t) \cdot \frac{1.0 - sp^{\mathrm{L}}}{sp^{\mathrm{L}}}. \tag{5.13}$$

We will show that, by (5.13), regardless of whether $d_{i_2}(t) < d^{\text{big}}\big(\bar{\mathbf{X}}^{\text{USE}}(t), t\big)$ or $d_{i_2}(t) \geq d^{\text{big}}\big(\bar{\mathbf{X}}^{\text{USE}}(t), t\big)$, we have $d_{i_1}(t) < \bar{d}_{i_2}\big(\bar{\mathbf{X}}^{\text{USE}}(t), t\big)$. Suppose we have $d_{i_2}(t) < d^{\text{big}}\big(\bar{\mathbf{X}}^{\text{USE}}(t), t\big)$. Recall that we have assumed the connected component of interest is as illustrated in Figure 5.15d. Because τ_{i_2} is incident on two edges in Figure 5.15d, by Lemmas 5.16 and 5.17, we have $\tau_{i_2} \in \tau^{\text{glob}}$. Because τ_{i_2} is matched to LITTLE CPU π_{j_1} in $\bar{\mathbb{M}}^{\text{USE}}(t)$ in Figure 5.15d, we have $\bar{x}^{\text{USE}}_{i_2,j_1}(t) = 1$ and $\pi_{j_1} \in \pi^{\text{LIT}}$. Thus,

$$\begin{aligned}
d_{i_1}(t) < {} & \{\text{Equation (5.13)}\} \\
& d^{\text{big}}\big(\bar{\mathbf{X}}^{\text{USE}}(t), t\big) \cdot \frac{1.0}{sp^{\mathrm{L}}} - d_{i_2}(t) \cdot \frac{1.0 - sp^{\mathrm{L}}}{sp^{\mathrm{L}}} \\
= {} & \big\{\text{Definition 5.8, } d_{i_2}(t) < d^{\text{big}}\big(\bar{\mathbf{X}}^{\text{USE}}(t), t\big), \tau_{i_2} \in \tau^{\text{glob}}, \bar{x}^{\text{USE}}_{i_2,j_1}(t) = 1, \text{ and } \pi_{j_1} \in \pi^{\text{LIT}}\big\} \\
& \bar{d}_{i_2}\big(\bar{\mathbf{X}}^{\text{USE}}(t), t\big).
\end{aligned}$$

Suppose we instead have $d_{i_2}(t) \geq d^{\text{big}}\big(\bar{\mathbf{X}}^{\text{USE}}(t), t\big)$. We have

$$\begin{aligned}
d_{i_1}(t) < {} & \{\text{Equation (5.12)}\} \\
& d^{\text{big}}\big(\bar{\mathbf{X}}^{\text{USE}}(t), t\big) \cdot \frac{1.0}{sp^{\mathrm{L}}} - d_{i_2}(t) \cdot \frac{1.0 - sp^{\mathrm{L}}}{sp^{\mathrm{L}}} \\
\leq {} & d_{i_2}(t) \cdot \frac{1.0}{sp^{\mathrm{L}}} - d_{i_2}(t) \cdot \frac{1.0 - sp^{\mathrm{L}}}{sp^{\mathrm{L}}} \\
= {} & d_{i_2}(t) \\
\leq {} & \{\text{Lemma 5.4}\} \\
& \bar{d}_{i_2}\big(\bar{\mathbf{X}}^{\text{USE}}(t), t\big).
\end{aligned}$$

Regardless of whether $d_{i_2}(t) < d^{\text{big}}\big(\bar{\mathbf{X}}^{\text{USE}}(t), t\big)$ or $d_{i_2}(t) \geq d^{\text{big}}\big(\bar{\mathbf{X}}^{\text{USE}}(t), t\big)$, we have $d_{i_1}(t) < \bar{d}_{i_2}\big(\bar{\mathbf{X}}^{\text{USE}}(t), t\big)$. This contradicts USE 1. ♦

All cases contradict USE 1. Thus, either $\bar{\mathbf{X}}^{\text{USE}}(t)$ is not a configuration selected by Ufm-SC-EDF or configuration $\bar{\mathbf{X}}^{\text{opt}}(t)$ with higher objective function value does not exist. Because the configuration chosen by Ufm-SC-EDF is optimal, Ufm-SC-EDF is a special case of Unr-WC. □

5.3.4 ACS Conditions

This subsection derives conditions maintained by our patched ACS that, on our platform, are equivalent to (3.39)-(3.41). We modify the ACS to maintain conditions (3.39)-(3.41) in order to guarantee bounded response times (by Theorem 3.36). Direct implementation of (3.39)-(3.41), which can be solved as a linear program, is impractical in the Linux kernel. Similarly to how Ufm-SC-EDF is a special case of Unr-WC, a special case of (3.39)-(3.41) can be implemented on our assumed platform. Note that in our implementation, the value used for $s\ell$ in (3.39) is

$$s\ell = 1.0 - \frac{\texttt{sched\_rt\_runtime\_us}}{\texttt{sched\_rt\_period\_us}}.$$

Recall from Section 4.1.5 that `sched_rt_runtime_us`/`sched_rt_period_us`, when the ACS is enabled, represents the fraction of CPU capacity permitted to be consumed by `SCHED_DEADLINE` tasks. The value $1.0 - s\ell$, which represents the fraction of each CPU's speed necessary for the task system to remain feasible (recall Definition 3.8), is a natural analogue of `sched_rt_runtime_us`/`sched_rt_period_us`.

The special case of (3.39)-(3.41) that we implement depends on the following definition.

▽ **Definition 5.14.** For task $\tau_i \in \tau^{\text{glob}}$, let

$$u_i^{\text{big}} \triangleq \begin{cases} \frac{u_i - (1.0 - s\ell) \cdot sp^{\text{L}}}{1.0 - sp^{\text{L}}} & u_i > (1.0 - s\ell) \cdot sp^{\text{L}} \\ 0 & u_i \leq (1.0 - s\ell) \cdot sp^{\text{L}} \end{cases}. \qquad \triangle$$

Note that a task τ_i with $u_i > (1.0 - s\ell) \cdot sp^{\text{L}}$ has higher bandwidth than the capacity provided by any LITTLE CPU. Task τ_i would be starved if scheduled exclusively on LITTLE CPUs. For such a task τ_i, u_i^{big}

roughly corresponds with the minimum component of task τ_i's bandwidth u_i that, in the long-term, must be serviced by big CPUs.

We will show that (3.39)-(3.41) are equivalent to the following.

$$\forall \tau_i \in \tau_{\text{act}}^{\text{big}}(t) \cup \tau_{\text{act}}^{\text{glob}}(t) : \; u_i \leq 1.0 - s\ell \tag{5.14}$$

$$\forall \tau_i \in \tau_{\text{act}}^{\text{LIT}}(t) : \; u_i \leq (1.0 - s\ell) \cdot sp^{\text{L}} \tag{5.15}$$

$$\left(\sum_{\tau_i \in \tau_{\text{act}}^{\text{big}}(t)} u_i \right) + \left(\sum_{\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)} u_i^{\text{big}} \right) \leq (1.0 - s\ell) \cdot m^{\text{big}} \tag{5.16}$$

$$\sum_{\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)} u_i \leq (1.0 - s\ell) \cdot m^{\text{LIT}} \cdot sp^{\text{L}} \tag{5.17}$$

$$\sum_{\tau_i \in \tau_{\text{act}}(t)} u_i \leq (1.0 - s\ell) \cdot (m^{\text{big}} + m^{\text{LIT}} \cdot sp^{\text{L}}) \tag{5.18}$$

The above are the conditions that will be maintained in our patched ACS. We will briefly discuss how the ACS is patched in Section 5.3.5.3.

▷ **Lemma 5.23.** For any time t, if there exists $\mathbf{X} \in \mathbb{R}_{\geq 0}^{n \times m}$ such that (3.39)-(3.41) are true, then (5.14)-(5.18) are true. ◁

Proof. Consider any $\mathbf{X} \in \mathbb{R}_{\geq 0}^{n \times m}$. We consider each of (5.14)-(5.18).

For (5.14), we prove the contrapositive: if (5.14) is false, then at least one of (3.39)-(3.41) is false. As guaranteed by the negation of (5.14), let task $\tau_i \in \tau_{\text{act}}^{\text{big}}(t) \cup \tau_{\text{act}}^{\text{glob}}(t)$ be such that $u_i > 1.0 - s\ell$. Assume that (3.40) is true of $\mathbf{X}$, as otherwise one of (3.39)-(3.41) is false, which is our proof obligation when considering (5.14). We have

$$\begin{aligned}
&\sum_{\pi_j \in \pi} (1.0 - s\ell) \cdot sp^{i,j} \cdot x_{i,j} \\
\leq \; &\left\{ \text{By Definition 5.2 and (5.2)-(5.7), } sp^{i,j} \leq 1.0 \right\} \\
&\sum_{\pi_j \in \pi} (1.0 - s\ell) \cdot x_{i,j} \\
= \; &(1.0 - s\ell) \sum_{\pi_j \in \pi} x_{i,j}
\end{aligned}$$

$$
\begin{aligned}
&\leq \{\text{Equation (3.40)}\} \\
&\quad 1.0 - s\ell \\
&< u_i.
\end{aligned}
$$

The above is the negation of (3.39). Thus, if (5.14) is false, at least one of (3.39)-(3.41) is false. This is the contrapositive.

We also prove the contrapositive for (5.15): if (5.15) is false, then at least one of (3.39)-(3.41) is false. As guaranteed by the negation of (5.15), let task $\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)$ be such that $u_i > (1.0 - s\ell) \cdot sp^{\text{L}}$. Assume that (3.40) is true, as otherwise one of (3.39)-(3.41) is false, which is our proof obligation when considering (5.15). We have

$$
\begin{aligned}
&\sum_{\pi_j \in \pi} (1.0 - s\ell) \cdot sp^{i,j} \cdot x_{i,j} \\
&= \left(\sum_{\pi_j \in \pi^{\text{big}}} (1.0 - s\ell) \cdot sp^{i,j} \cdot x_{i,j} \right) + \left(\sum_{\pi_j \in \pi^{\text{LIT}}} (1.0 - s\ell) \cdot sp^{i,j} \cdot x_{i,j} \right) \\
&= \left\{\text{By (5.6) and } \tau_i \in \tau_{\text{act}}^{\text{LIT}}(t) \subseteq \tau^{\text{LIT}} \wedge \pi_j \in \pi^{\text{big}},\ sp^{i,j} = 0\right\} \\
&\quad \left(\sum_{\pi_j \in \pi^{\text{big}}} (1.0 - s\ell) \cdot 0 \cdot x_{i,j} \right) + \left(\sum_{\pi_j \in \pi^{\text{LIT}}} (1.0 - s\ell) \cdot sp^{i,j} \cdot x_{i,j} \right) \\
&= \sum_{\pi_j \in \pi^{\text{LIT}}} (1.0 - s\ell) \cdot sp^{i,j} \cdot x_{i,j} \\
&= \left\{\text{By (5.7 and } \tau_i \in \tau_{\text{act}}^{\text{LIT}}(t) \subseteq \tau^{\text{LIT}} \wedge \pi_j \in \pi^{\text{LIT}},\ sp^{i,j} = sp^{\text{L}}\right\} \\
&\quad \sum_{\pi_j \in \pi^{\text{LIT}}} (1.0 - s\ell) \cdot sp^{\text{L}} \cdot x_{i,j} \\
&= (1.0 - s\ell) \cdot sp^{\text{L}} \cdot \sum_{\pi_j \in \pi^{\text{LIT}}} x_{i,j} \\
&\leq (1.0 - s\ell) \cdot sp^{\text{L}} \cdot \sum_{\pi_j \in \pi} x_{i,j} \\
&\leq \{\text{Equation (3.40)}\} \\
&\quad (1.0 - s\ell) \cdot sp^{\text{L}} \\
&< u_i.
\end{aligned}
$$

The above is the negation of (3.39). Thus, if (5.15) is false, at least one of (3.39)-(3.41) is false. This is the contrapositive.

For (5.16), consider any task $\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)$. We have

$$\left.\begin{aligned}
&\sum_{\pi_j \in \pi^{\text{big}}} (1.0 - s\ell) \cdot x_{i,j} \\
&= \Big\{\text{By (5.2) and } \tau_i \in \tau_{\text{act}}^{\text{glob}}(t) \subseteq \tau^{\text{glob}} \wedge \pi_j \in \pi^{\text{big}},\ sp^{i,j} = 1.0\Big\} \\
&\quad \sum_{\pi_j \in \pi^{\text{big}}} (1.0 - s\ell) \cdot sp^{i,j} \cdot x_{i,j} \\
&\geq \{\text{Equation (3.39)}\} \\
&\quad u_i - \sum_{\pi_j \in \pi^{\text{LIT}}} (1.0 - s\ell) \cdot sp^{i,j} \cdot x_{i,j} \\
&= \Big\{\text{By (5.3) and } \tau_i \in \tau_{\text{act}}^{\text{glob}}(t) \subseteq \tau^{\text{glob}} \wedge \pi_j \in \pi^{\text{LIT}},\ sp^{i,j} = sp^{\text{L}}\Big\} \\
&\quad u_i - \sum_{\pi_j \in \pi^{\text{LIT}}} (1.0 - s\ell) \cdot sp^{\text{L}} \cdot x_{i,j} \\
&= u_i - (1.0 - s\ell) \cdot sp^{\text{L}} \cdot \sum_{\pi_j \in \pi^{\text{LIT}}} x_{i,j} \\
&\geq \{\text{Equation (3.40)}\} \\
&\quad u_i - (1.0 - s\ell) \cdot sp^{\text{L}} \cdot \left(1.0 - \sum_{\pi_j \in \pi^{\text{big}}} x_{i,j}\right) \\
&= u_i - (1.0 - s\ell) \cdot sp^{\text{L}} + (1.0 - s\ell) \cdot sp^{\text{L}} \cdot \sum_{\pi_j \in \pi^{\text{big}}} x_{i,j} \\
&= u_i - (1.0 - s\ell) \cdot sp^{\text{L}} + sp^{\text{L}} \cdot \sum_{\pi_j \in \pi^{\text{big}}} (1.0 - s\ell) \cdot x_{i,j}
\end{aligned}\right\} \tag{5.19}$$

Subtracting both sides of (5.19) by $sp^{\text{L}} \cdot \sum_{\pi_j \in \pi^{\text{big}}} (1.0 - s\ell) \cdot x_{i,j}$ and then dividing both sides by $1.0 - sp^{\text{L}}$ yields

$$\left.\begin{aligned}
\sum_{\pi_j \in \pi^{\text{big}}} (1.0 - s\ell) \cdot x_{i,j} &\geq \frac{u_i - (1.0 - s\ell) \cdot sp^{\text{L}}}{1.0 - sp^{\text{L}}} \\
&\geq \{\text{Definition 5.14}\} \\
&\quad u_i^{\text{big}}.
\end{aligned}\right\} \tag{5.20}$$

We have

$$\left(\sum_{\tau_i \in \tau_{\text{act}}^{\text{big}}(t)} u_i\right) + \left(\sum_{\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)} u_i^{\text{big}}\right)$$

$\leq$ {Equation (3.39)}

$$\left(\sum_{\tau_i \in \tau_{\text{act}}^{\text{big}}(t)} \sum_{\pi_j \in \pi} (1.0 - s\ell) \cdot sp^{i,j} \cdot x_{i,j}\right) + \left(\sum_{\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)} u_i^{\text{big}}\right)$$

$$= \left(\sum_{\tau_i \in \tau_{\text{act}}^{\text{big}}(t)} \left[\sum_{\pi_j \in \pi^{\text{big}}} (1.0 - s\ell) \cdot sp^{i,j} \cdot x_{i,j} + \sum_{\pi_j \in \pi^{\text{LIT}}} (1.0 - s\ell) \cdot sp^{i,j} \cdot x_{i,j}\right]\right) + \left(\sum_{\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)} u_i^{\text{big}}\right)$$

$$= \left\{\text{By (5.4) and } \tau_i \in \tau_{\text{act}}^{\text{big}}(t) \subseteq \tau^{\text{big}} \wedge \pi_j \in \pi^{\text{big}},\ sp^{i,j} = 1.0\right\}$$

$$\left(\sum_{\tau_i \in \tau_{\text{act}}^{\text{big}}(t)} \left[\sum_{\pi_j \in \pi^{\text{big}}} (1.0 - s\ell) \cdot x_{i,j} + \sum_{\pi_j \in \pi^{\text{LIT}}} (1.0 - s\ell) \cdot sp^{i,j} \cdot x_{i,j}\right]\right) + \left(\sum_{\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)} u_i^{\text{big}}\right)$$

$$= \left\{\text{By (5.5) and } \tau_i \in \tau_{\text{act}}^{\text{big}}(t) \subseteq \tau^{\text{big}} \wedge \pi_j \in \pi^{\text{LIT}},\ sp^{i,j} = 0\right\}$$

$$\left(\sum_{\tau_i \in \tau_{\text{act}}^{\text{big}}(t)} \left[\sum_{\pi_j \in \pi^{\text{big}}} (1.0 - s\ell) \cdot x_{i,j} + \sum_{\pi_j \in \pi^{\text{LIT}}} (1.0 - s\ell) \cdot 0 \cdot x_{i,j}\right]\right) + \left(\sum_{\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)} u_i^{\text{big}}\right)$$

$$= \left(\sum_{\tau_i \in \tau_{\text{act}}^{\text{big}}(t)} \sum_{\pi_j \in \pi^{\text{big}}} (1.0 - s\ell) \cdot x_{i,j}\right) + \left(\sum_{\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)} u_i^{\text{big}}\right)$$

$\leq$ {Equation (5.20)}

$$\left(\sum_{\tau_i \in \tau_{\text{act}}^{\text{big}}(t)} \sum_{\pi_j \in \pi^{\text{big}}} (1.0 - s\ell) \cdot x_{i,j}\right) + \left(\sum_{\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)} \sum_{\pi_j \in \pi^{\text{big}}} (1.0 - s\ell) \cdot x_{i,j}\right)$$

$$= \left(\sum_{\pi_j \in \pi^{\text{big}}} \sum_{\tau_i \in \tau_{\text{act}}^{\text{big}}(t)} (1.0 - s\ell) \cdot x_{i,j}\right) + \left(\sum_{\pi_j \in \pi^{\text{big}}} \sum_{\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)} (1.0 - s\ell) \cdot x_{i,j}\right)$$

$$= (1.0 - s\ell) \sum_{\pi_j \in \pi^{\text{big}}} \sum_{\tau_i \in \tau_{\text{act}}^{\text{big}}(t) \cup \tau_{\text{act}}^{\text{glob}}(t)} x_{i,j}$$

$\leq$ $\left\{\tau_{\text{act}}^{\text{big}}(t) \cup \tau_{\text{act}}^{\text{glob}}(t) \subseteq \tau_{\text{act}}(t),\ s\ell < 1.0 \text{ (by Definition 3.8), and } \mathbf{X} \in \mathbb{R}_{\geq 0}^{n \times m}\right\}$

$$(1.0 - s\ell) \sum_{\pi_j \in \pi^{\text{big}}} \sum_{\tau_i \in \tau_{\text{act}}(t)} x_{i,j}$$

$$\leq \{\text{Equation (3.41)}\}$$

$$(1.0 - s\ell) \sum_{\pi_j \in \pi^{\text{big}}} 1.0$$

$$= (1.0 - s\ell) \cdot m^{\text{big}}.$$

This is (5.16).

For (5.17), we have

$$\begin{aligned}
\sum_{\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)} u_i &\leq \{\text{Equation (3.39)}\} \\
&\quad \sum_{\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)} \sum_{\pi_j \in \pi} (1.0 - s\ell) \cdot sp^{i,j} \cdot x_{i,j} \\
&= (1.0 - s\ell) \sum_{\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)} \sum_{\pi_j \in \pi} sp^{i,j} \cdot x_{i,j} \\
&= (1.0 - s\ell) \sum_{\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)} \left(\sum_{\pi_j \in \pi^{\text{LIT}}} sp^{i,j} \cdot x_{i,j} + \sum_{\pi_j \in \pi^{\text{big}}} sp^{i,j} \cdot x_{i,j} \right) \\
&= \left\{\text{By (5.6) and } \tau_i \in \tau_{\text{act}}^{\text{LIT}}(t) \subseteq \tau^{\text{LIT}} \wedge \pi_j \in \pi^{\text{big}},\ sp^{i,j} = 0\right\} \\
&\quad (1.0 - s\ell) \sum_{\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)} \sum_{\pi_j \in \pi^{\text{LIT}}} sp^{i,j} \cdot x_{i,j} \\
&= \left\{\text{By (5.7) and } \tau_i \in \tau_{\text{act}}^{\text{LIT}}(t) \subseteq \tau^{\text{LIT}} \wedge \pi_j \in \pi^{\text{LIT}},\ sp^{i,j} = sp^{\text{L}}\right\} \\
&\quad (1.0 - s\ell) \sum_{\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)} \sum_{\pi_j \in \pi^{\text{LIT}}} sp^{\text{L}} \cdot x_{i,j} \\
&= (1.0 - s\ell) \cdot sp^{\text{L}} \cdot \sum_{\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)} \sum_{\pi_j \in \pi^{\text{LIT}}} x_{i,j} \\
&= (1.0 - s\ell) \cdot sp^{\text{L}} \cdot \sum_{\pi_j \in \pi^{\text{LIT}}} \sum_{\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)} x_{i,j} \\
&\leq \{\text{Equation (3.41)}\} \\
&\quad (1.0 - s\ell) \cdot sp^{\text{L}} \cdot \sum_{\pi_j \in \pi^{\text{LIT}}} 1.0 \\
&= (1.0 - s\ell) \cdot sp^{\text{L}} \cdot m^{\text{LIT}}.
\end{aligned}$$

This is (5.17).

For (5.18), we have

$$
\begin{aligned}
&\sum_{\tau_i \in \tau_{\text{act}}(t)} u_i \\
&\leq \{\text{Equation (3.39)}\} \\
&\quad \sum_{\tau_i \in \tau_{\text{act}}(t)} \sum_{\pi_j \in \pi} (1.0 - s\ell) \cdot sp^{i,j} \cdot x_{i,j} \\
&= (1.0 - s\ell) \sum_{\tau_i \in \tau_{\text{act}}(t)} \sum_{\pi_j \in \pi} sp^{i,j} \cdot x_{i,j} \\
&= (1.0 - s\ell) \sum_{\pi_j \in \pi} \sum_{\tau_i \in \tau_{\text{act}}(t)} sp^{i,j} \cdot x_{i,j} \\
&= (1.0 - s\ell) \left(\sum_{\pi_j \in \pi^{\text{big}}} \sum_{\tau_i \in \tau_{\text{act}}(t)} sp^{i,j} \cdot x_{i,j} + \sum_{\pi_j \in \pi^{\text{LIT}}} \sum_{\tau_i \in \tau_{\text{act}}(t)} sp^{i,j} \cdot x_{i,j} \right) \\
&= (1.0 - s\ell) \left(\sum_{\pi_j \in \pi^{\text{big}}} \begin{bmatrix} \sum_{\tau_i \in \tau_{\text{act}}^{\text{big}}(t)} sp^{i,j} \cdot x_{i,j} \\ + \sum_{\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)} sp^{i,j} \cdot x_{i,j} \\ + \sum_{\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)} sp^{i,j} \cdot x_{i,j} \end{bmatrix} + \sum_{\pi_j \in \pi^{\text{LIT}}} \begin{bmatrix} \sum_{\tau_i \in \tau_{\text{act}}^{\text{big}}(t)} sp^{i,j} \cdot x_{i,j} \\ + \sum_{\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)} sp^{i,j} \cdot x_{i,j} \\ + \sum_{\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)} sp^{i,j} \cdot x_{i,j} \end{bmatrix} \right) \\
&= \{\text{Equations (5.2), (5.4), and (5.6)}\} \\
&\quad (1.0 - s\ell) \left(\sum_{\pi_j \in \pi^{\text{big}}} \begin{bmatrix} \sum_{\tau_i \in \tau_{\text{act}}^{\text{big}}(t)} x_{i,j} \\ + \sum_{\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)} 0 \cdot x_{i,j} \\ + \sum_{\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)} x_{i,j} \end{bmatrix} + \sum_{\pi_j \in \pi^{\text{LIT}}} \begin{bmatrix} \sum_{\tau_i \in \tau_{\text{act}}^{\text{big}}(t)} sp^{i,j} \cdot x_{i,j} \\ + \sum_{\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)} sp^{i,j} \cdot x_{i,j} \\ + \sum_{\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)} sp^{i,j} \cdot x_{i,j} \end{bmatrix} \right) \\
&= (1.0 - s\ell) \left(\sum_{\pi_j \in \pi^{\text{big}}} \begin{bmatrix} \sum_{\tau_i \in \tau_{\text{act}}^{\text{big}}(t)} x_{i,j} \\ + \sum_{\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)} x_{i,j} \end{bmatrix} + \sum_{\pi_j \in \pi^{\text{LIT}}} \begin{bmatrix} \sum_{\tau_i \in \tau_{\text{act}}^{\text{big}}(t)} sp^{i,j} \cdot x_{i,j} \\ + \sum_{\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)} sp^{i,j} \cdot x_{i,j} \\ + \sum_{\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)} sp^{i,j} \cdot x_{i,j} \end{bmatrix} \right) \\
&\leq \left\{ \tau_{\text{act}}(t) = \tau_{\text{act}}^{\text{big}}(t) \cup \tau_{\text{act}}^{\text{LIT}}(t) \cup \tau_{\text{act}}^{\text{glob}}(t) \text{ and } \mathbf{X} \in \mathbb{R}_{\geq 0}^{n \times m} \right\} \\
&\quad (1.0 - s\ell) \left(\sum_{\pi_j \in \pi^{\text{big}}} \sum_{\tau_i \in \tau_{\text{act}}(t)} x_{i,j} + \sum_{\pi_j \in \pi^{\text{LIT}}} \begin{bmatrix} \sum_{\tau_i \in \tau_{\text{act}}^{\text{big}}(t)} sp^{i,j} \cdot x_{i,j} \\ + \sum_{\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)} sp^{i,j} \cdot x_{i,j} \\ + \sum_{\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)} sp^{i,j} \cdot x_{i,j} \end{bmatrix} \right) \\
&= \{\text{Equations (5.3), (5.5), and (5.7)}\}
\end{aligned}
$$

$$
\begin{aligned}
&(1.0 - s\ell) \left(\sum_{\pi_j \in \pi^{\text{big}}} \sum_{\tau_i \in \tau_{\text{act}}(t)} x_{i,j} + \sum_{\pi_j \in \pi^{\text{LIT}}} \begin{bmatrix} \sum_{\tau_i \in \tau_{\text{act}}^{\text{big}}(t)} 0 \cdot x_{i,j} \\ + \sum_{\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)} sp^{\text{L}} \cdot x_{i,j} \\ + \sum_{\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)} sp^{\text{L}} \cdot x_{i,j} \end{bmatrix} \right) \\
&\leq \left\{ \tau_{\text{act}}(t) = \tau_{\text{act}}^{\text{big}}(t) \cup \tau_{\text{act}}^{\text{LIT}}(t) \cup \tau_{\text{act}}^{\text{glob}}(t) \text{ and } \mathbf{X} \in \mathbb{R}_{\geq 0}^{n \times m} \right\} \\
&\quad (1.0 - s\ell) \left(\sum_{\pi_j \in \pi^{\text{big}}} \sum_{\tau_i \in \tau_{\text{act}}(t)} x_{i,j} + \sum_{\pi_j \in \pi^{\text{LIT}}} \sum_{\tau_i \in \tau_{\text{act}}(t)} sp^{\text{L}} \cdot x_{i,j} \right) \\
&= (1.0 - s\ell) \left(\sum_{\pi_j \in \pi^{\text{big}}} \sum_{\tau_i \in \tau_{\text{act}}(t)} x_{i,j} + sp^{\text{L}} \cdot \sum_{\pi_j \in \pi^{\text{LIT}}} \sum_{\tau_i \in \tau_{\text{act}}(t)} x_{i,j} \right) \\
&\leq \{\text{Equation (3.41)}\} \\
&\quad (1.0 - s\ell) \left(\sum_{\pi_j \in \pi^{\text{big}}} 1.0 + sp^{\text{L}} \cdot \sum_{\pi_j \in \pi^{\text{LIT}}} 1.0 \right) \\
&= (1.0 - s\ell) \left(m^{\text{big}} + sp^{\text{L}} \cdot m^{\text{LIT}} \right).
\end{aligned}
$$

This is (5.18).

All of (5.14)-(5.18) are implied by (3.39)-(3.41). This completes the proof. □

Lemma 5.23 proves one direction of the equivalence between (3.39)-(3.41) and (5.14)-(5.18). We prove the other direction by providing an algorithm which, if (5.14)-(5.18) are true at time t, constructs an $\mathbf{X}$ that satisfies (3.39)-(3.41). The high-level steps of this algorithm are as follows.

Step 1. Initialize the elements of $\mathbf{X}$ to 0.

Step 2. Allocate u_i of capacity to each $\tau_i \in \tau_{\text{act}}^{\text{big}}(t)$ from the CPUs in π^{big}.

Step 3. Allocate u_i^{big} of capacity to each $\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)$ from the CPUs in π^{big}.

Step 4. Allocate u_i of capacity to each $\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)$ from the CPUs in π^{LIT}.

Step 5. Allocate $u_i - u_i^{\text{big}}$ of capacity to each $\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)$ from any of the CPUs in π.

Example 5.8 illustrates this algorithm.

▼ **Example 5.8.** Consider a system with $m^{\text{big}} = 3$, $m^{\text{LIT}} = 3$, $s\ell = 0.05$, and $sp^{\text{L}} = 0.6$. In Figure 5.16, we represent the capacity (scaled by $1.0 - s\ell$) of each CPU with a rectangle. Each rectangle has a height

equal to its corresponding CPU's scaled capacity (*i.e.*, $1.0 - s\ell$ for the CPUs in π^{big} and $(1.0 - s\ell) \cdot sp^{\text{L}}$ for the CPUs in π^{LIT}), and width 1.0, which reflects the fraction of this CPU's capacity that has been allocated to tasks. Initially, all rectangles are shaded to reflect that no allocations of capacity have been given to tasks in Step 1. Subsequent steps of the algorithm will allocate blocks of these rectangles to tasks. The value of $x_{i,j}$ set by the algorithm is the total width of blocks from CPU π_j's rectangle allocated to task τ_i after all steps have completed.

At time t, suppose the active tasks are such that $\tau^{\text{big}}_{\text{act}}(t) = \{\tau_1, \tau_2\}$, $\tau^{\text{glob}}_{\text{act}}(t) = \{\tau_3, \tau_4, \tau_5, \tau_6\}$, and $\tau^{\text{LIT}}_{\text{act}}(t) = \{\tau_7, \tau_8, \tau_9\}$. Let $u_1 = 0.3$, $u_2 = 0.2$, $u_3 = 0.7$, $u_4 = 0.8$, $u_5 = 0.7$, $u_6 = 0.5$, $u_7 = 0.4$, $u_8 = 0.3$, and $u_9 = 0.5$.

Step 2 is illustrated in Figure 5.17. Each $\tau_i \in \tau^{\text{big}}_{\text{act}}(t)$ is allocated u_i of capacity. Blocks of π_1, π_2, and π_3 in π^{big} are provided to tasks τ_1 and τ_2 in $\tau^{\text{big}}_{\text{act}}(t)$. For example, to allocate $u_1 = 0.3$ of capacity to task τ_1, a block of CPU π_1 is allocated to τ_1 with width $\frac{u_1}{1.0-s\ell} = \frac{0.3}{1.0-0.05} = 0.316$ such that the block has area $0.316 \cdot (1.0 - s\ell) = 0.316 \cdot (1.0 - 0.05) = 0.3 = u_1$.[4] $x_{1,1}$ is set to 0.316, matching the width of this block.

Step 3 is illustrated in Figure 5.18. Each $\tau_i \in \tau^{\text{glob}}_{\text{act}}(t)$ is allocated u^{big}_i of capacity. Blocks of π_1, π_2, and π_3 in π^{big} are provided to tasks τ_3, τ_4, and τ_5. For example, to allocate $u^{\text{big}}_4 = \frac{u_4 - (1.0-s\ell)\cdot sp^{\text{L}}}{1.0-sp^{\text{L}}} = \frac{0.8-0.95\cdot 0.6}{1.0-0.6} = 0.576$ of capacity to task $\tau_4 \in \tau^{\text{glob}}_{\text{act}}(t)$, a block of π_1 with width 0.132 and a block of π_2 with width 0.474 are allocated to task τ_4. The total capacity is $0.132 \cdot (1.0 - s\ell) + 0.474 \cdot (1.0 - s\ell) = 0.132 \times 0.95 + 0.474 \cdot 0.95 = 0.576 = u^{\text{big}}_4$. $x_{4,1}$ is set to 0.132 and $x_{4,2}$ is set to 0.474. Task $\tau_6 \in \tau^{\text{glob}}_{\text{act}}(t)$ has $u_6 = 0.5 < 0.95 \cdot 0.6 = (1.0 - s\ell) \cdot sp^{\text{L}}$. Thus, τ_6 has $u^{\text{big}}_6 = 0$, and τ_6 is not allocated any blocks in Step 3.

Step 4 is illustrated in Figure 5.19. Each $\tau_i \in \tau^{\text{LIT}}_{\text{act}}(t)$ is allocated u_i of capacity. Blocks of π_4, π_5, and π_6 in π^{LIT} are provided to tasks τ_7, τ_8, and τ_9 in $\tau^{\text{LIT}}_{\text{act}}(t)$. For example, to allocate $u_8 = 0.3$ of capacity to task $\tau_8 \in \tau^{\text{LIT}}_{\text{act}}(t)$, a block of π_5 with width 0.228 and a block of π_6 with width 0.298 are allocated to task τ_8. The total area is $0.228 \cdot (1.0 - s\ell) \cdot sp^{\text{L}} + 0.298 \cdot (1.0 - s\ell) \cdot sp^{\text{L}} = 0.228 \cdot 0.95 \cdot 0.6 + 0.298 \cdot 0.95 \cdot 0.6 = 0.3 - u_8$. $x_{8,5}$ is set to 0.228 and $x_{8,6}$ is set to 0.298.

Step 5 is illustrated in Figure 5.20. Each $\tau_i \in \tau^{\text{glob}}_{\text{act}}(t)$ is allocated its remaining $u_i - u^{\text{big}}_i$ of capacity. Blocks from any CPU with remaining capacity are allocated to tasks τ_3, τ_4, τ_5, and τ_6. For example, task

[4]Note that all decimals in Example 5.8 have been truncated at three digits.

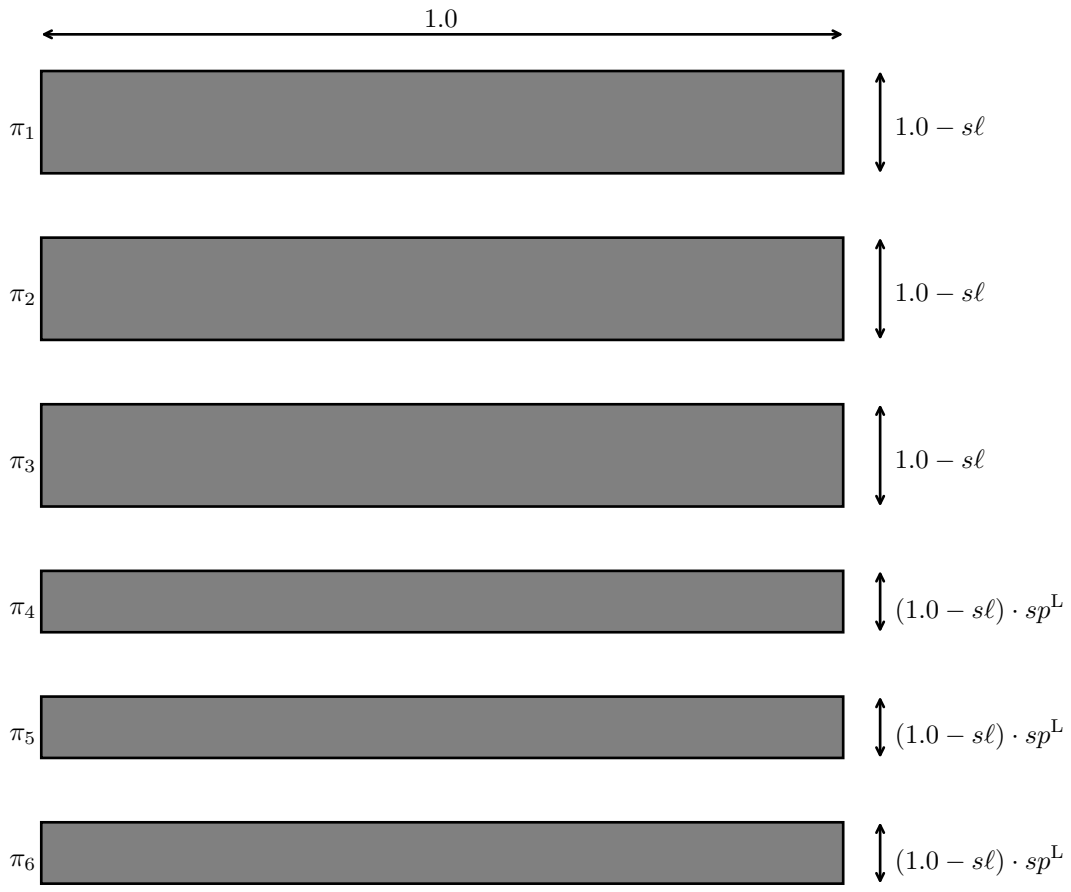


Figure 5.16: Step 1: initially, no tasks are allocated ($\mathbf{X} \leftarrow 0$).

$\tau_6 \in \tau_{\text{act}}^{\text{glob}}(t)$ has $u_6 - u_6^{\text{big}} = u_6 - 0 = u_6 = 0.5$ of remaining capacity that must be allocated in this Step 5. Task τ_6 is allocated blocks of width 0.158 on CPU π_3 and 0.614 on CPU π_4. This yields total area $0.158 \cdot (1.0 - s\ell) + 0.614 \cdot (1.0 - s\ell) \cdot sp^{\mathrm{L}} = 0.158 \cdot 0.95 + 0.614 \cdot 0.95 \cdot 0.6 = 0.5$.

After Step 5, $\mathbf{X}$ is set such that each task $\tau_i \in \tau_{\text{act}}(t)$ is allocated u_i of capacity. ▲

▷ **Lemma 5.24.** For any time t, if (5.14)-(5.18) are true, then there exists $\mathbf{X} \in \mathbb{R}_{\geq 0}^{n \times m}$ such that (3.39)-(3.41) are true. ◁

Proof. We prove that the algorithm discussed in Example 5.8 constructs $\mathbf{X} \in \mathbb{R}_{\geq 0}^{n \times m}$ such that (3.39)-(3.41) are true so long as (5.14)-(5.18) are true.

If the algorithm completes, the algorithm allocates u_i of area for each task $\tau_i \in \tau_{\text{act}}(t)$, which satisfies (3.39). The algorithm must also satisfy (3.41) if it completes because the total width of any

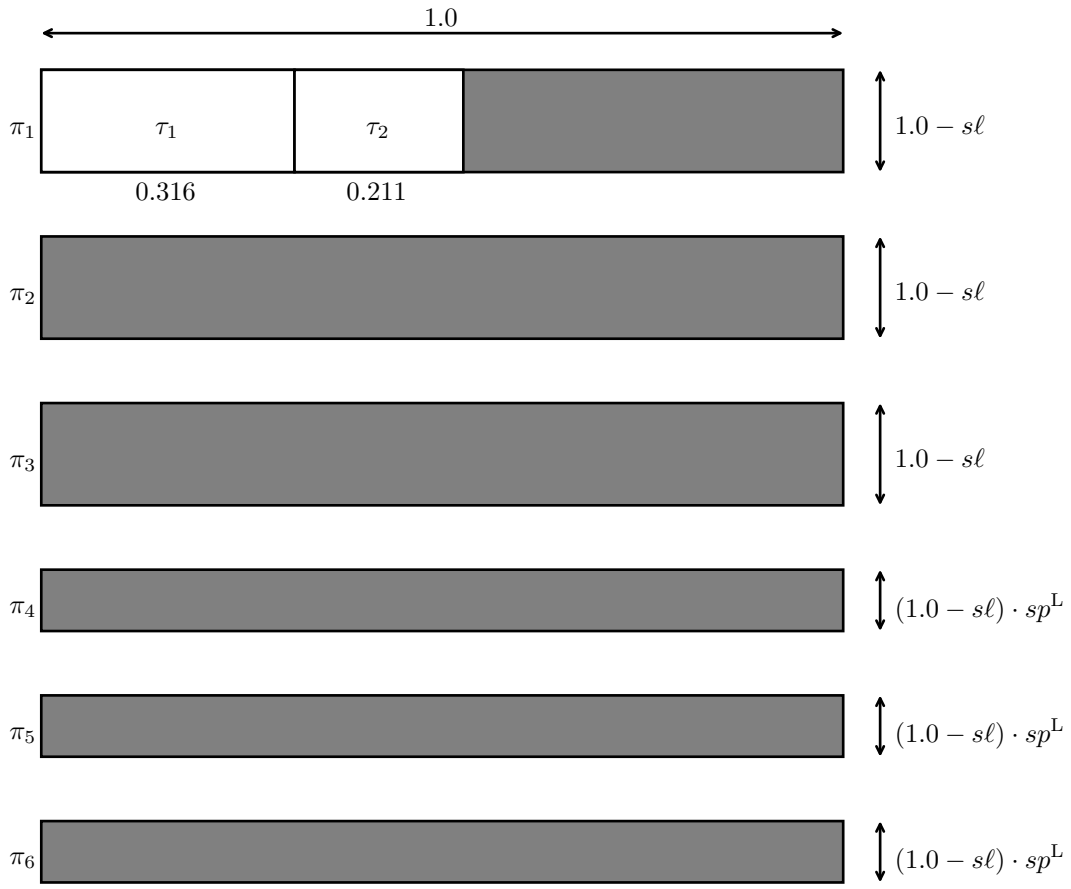


Figure 5.17: Step 2: each $\tau_i \in \tau_{\text{act}}^{\text{big}}(t)$ is allocated u_i of capacity in $\mathbf{X}$.

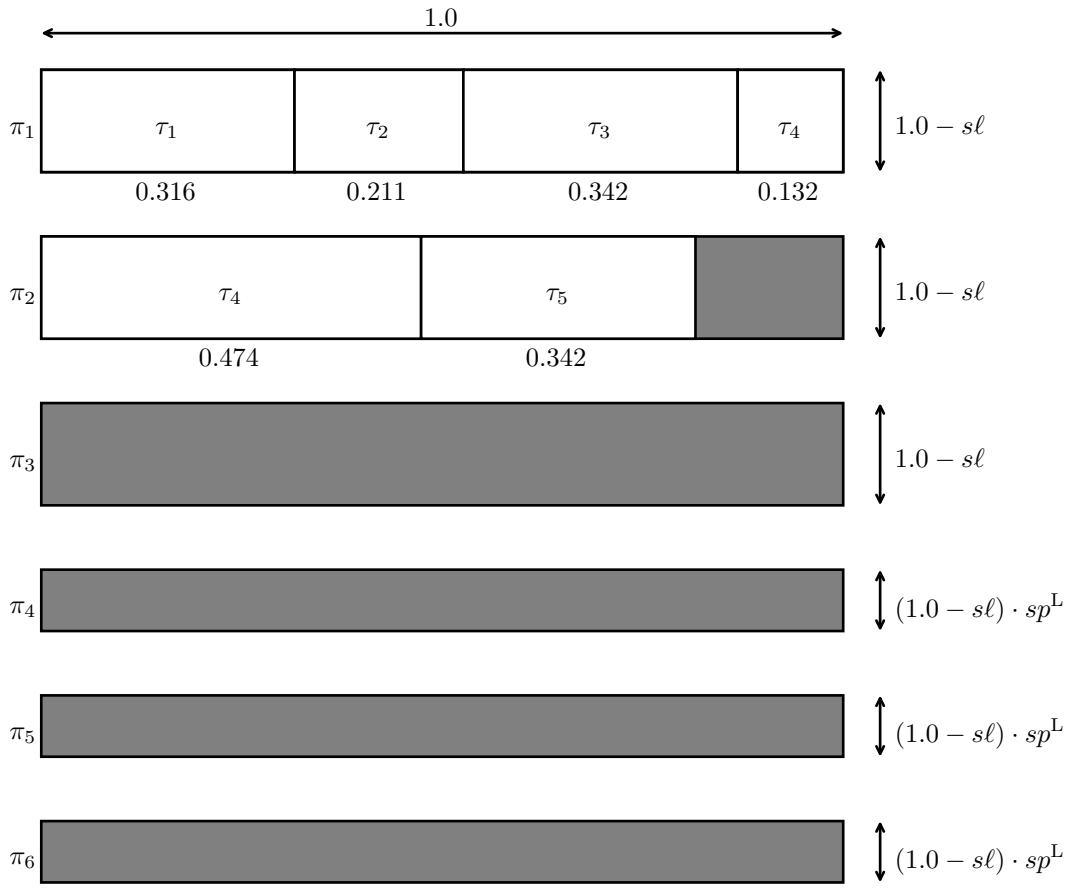


Figure 5.18: Step 3: each $\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)$ is allocated u_i^{big} of capacity in $\mathbf{X}$.

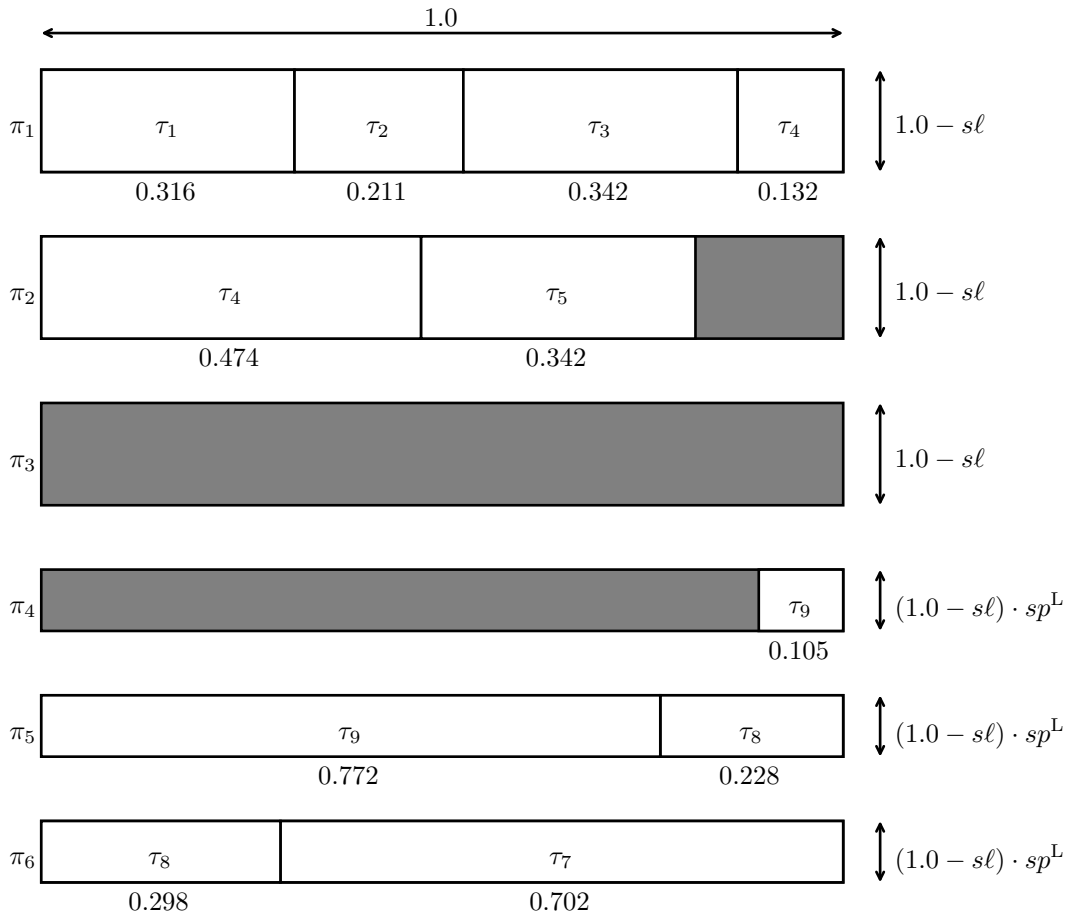


Figure 5.19: Step 4: each $\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)$ is allocated u_i of capacity in $\mathbf{X}$.

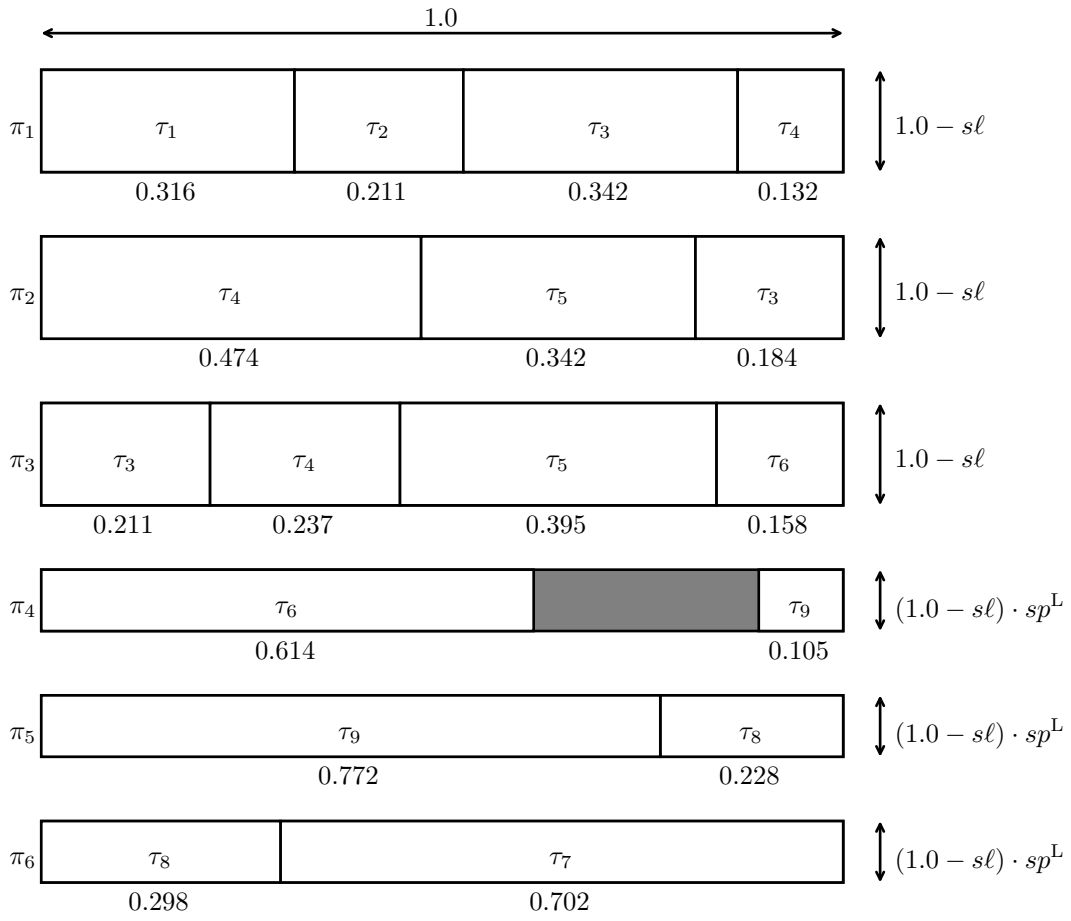


Figure 5.20: Step 5: each $\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)$ is allocated $u_i - u_i^{\text{big}}$ of capacity in $\mathbf{X}$.

rectangle belonging to a CPU is 1.0. The algorithm can only fail to complete if there is insufficient remaining capacity to allocate blocks as required by any step.

Recall that Step 2 allocates u_i of area for each task $\tau_i \in \tau_{\text{act}}^{\text{big}}(t)$ and Step 3 allocates u_i^{big} of area for each task $\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)$ from the CPUs in π^{big}. The total area of CPUs in π^{big} is $(1.0 - s\ell) \cdot m^{\text{big}}$. By (5.16), there is sufficient area for Step 2 and Step 3. Thus, Step 2 and Step 3 must complete.

Step 4 allocates u_i of area for each task $\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)$ from the CPUs in π^{LIT}. The total area of CPUs in π^{LIT} is $(1.0 - s\ell) \cdot sp^{\text{L}} \cdot m^{\text{LIT}}$. By (5.17), there is sufficient area for Step 4. Thus, Step 4 must complete.

Step 5 allocates $u_i - u_i^{\text{big}}$ of area for each task $\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)$ from any CPU with remaining capacity. The total area allocated from CPUs for tasks in $\tau_{\text{act}}^{\text{glob}}(t)$ in Step 3 and Step 5 is $\sum_{\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)} u_i^{\text{big}} + \left(u_i - u_i^{\text{big}}\right) = \sum_{\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)} u_i$. The total area allocated for tasks of $\tau_{\text{act}}^{\text{big}}(t)$ and $\tau_{\text{act}}^{\text{LIT}}(t)$ in Step 2 and Step 4 is $\sum_{\tau_i \in \tau_{\text{act}}^{\text{big}}(t) \cup \tau_{\text{act}}^{\text{LIT}}(t)} u_i$. The total area allocated to all tasks in $\tau_{\text{act}}(t) = \tau_{\text{act}}^{\text{big}}(t) \cup \tau_{\text{act}}^{\text{LIT}}(t) \cup \tau_{\text{act}}^{\text{glob}}(t)$ is $\sum_{\tau_i \in \tau_{\text{act}}(t)} u_i$. The total area to be allocated from CPUs is $(1.0 - s\ell) \cdot m^{\text{big}} + (1.0 - s\ell) \cdot sp^{\text{L}} \cdot m^{\text{LIT}} = (1.0 - s\ell) \cdot \left(m^{\text{big}} + m^{\text{LIT}} \cdot sp^{\text{L}}\right)$. By (5.18), there is sufficient area for Step 5. Thus, Step 5 must complete. Because Step 5 is the final step, the algorithm must complete. Because the algorithm completes, as discussed earlier in this proof, the algorithm satisfies (3.39) and (3.41).

It remains to prove that the algorithm satisfies (3.40) for each task in $\tau_{\text{act}}(t)$. For each task $\tau_i \in \tau_{\text{act}}^{\text{big}}(t)$, the algorithm only allocates area from CPUs in π^{big}, which have heights of $1.0 - s\ell$. Because the algorithm allocates u_i of area, the total width of blocks allocated to τ_i is $\frac{u_i}{1.0 - s\ell}$. By (5.14), the total width allocated to τ_i is at most 1.0. Because the widths of blocks correspond with the values of $x_{i,j}$ for each $\pi_j \in \pi$, (3.40) is satisfied for each task $\tau_i \in \tau_{\text{act}}^{\text{big}}(t)$.

For each task $\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)$, the algorithm only allocates area from CPUs in π^{LIT}, which have heights of $(1.0 - s\ell) \cdot sp^{\text{L}}$. Because the algorithm allocates u_i of area, the total width of blocks allocated to τ_i is $\frac{u_i}{(1.0 - s\ell) \cdot sp^{\text{L}}}$. By (5.15), the total width allocated to τ_i is at most 1.0. Because the widths of blocks correspond with the values of $x_{i,j}$ for each $\pi_j \in \pi$, (3.40) is satisfied for each task $\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)$.

For each task $\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)$, the algorithm allocates u_i of area, at least u_i^{big} of which must be allocated from CPUs in π^{big}. Because at least u_i^{big} of which must be allocated from CPUs in π^{big}, we have

$$\sum_{\pi_j \in \pi^{\text{big}}} (1.0 - s\ell) \cdot x_{i,j} \geq u_i^{\text{big}}$$

$$\geq \{\text{Definition 5.14}\}$$
$$\frac{u_i - (1.0 - s\ell) \cdot sp^{\mathrm{L}}}{1.0 - sp^{\mathrm{L}}}.$$

Multiplying both sides by $1.0 - sp^{\mathrm{L}}$ yields

$$\sum_{\pi_j \in \pi^{\text{big}}} (1.0 - s\ell) \cdot (1.0 - sp^{\mathrm{L}}) \cdot x_{i,j} \geq u_i - (1.0 - s\ell) \cdot sp^{\mathrm{L}}. \tag{5.21}$$

Because the algorithm allocates a total of u_i of area, we have

$$\begin{aligned}
u_i &= \sum_{\pi_j \in \pi^{\text{big}}} (1.0 - s\ell) \cdot x_{i,j} + \sum_{\pi_j \in \pi^{\text{LIT}}} (1.0 - s\ell) \cdot sp^{\mathrm{L}} \cdot x_{i,j} \\
&= \sum_{\pi_j \in \pi^{\text{big}}} (1.0 - s\ell) \cdot (1.0 - sp^{\mathrm{L}}) \cdot x_{i,j} + \sum_{\pi_j \in \pi^{\text{big}}} (1.0 - s\ell) \cdot sp^{\mathrm{L}} \cdot x_{i,j} \\
&\quad + \sum_{\pi_j \in \pi^{\text{LIT}}} (1.0 - s\ell) \cdot sp^{\mathrm{L}} \cdot x_{i,j} \\
&\leq \{\text{Equation (5.21)}\} \\
&\quad u_i - (1.0 - s\ell) \cdot sp^{\mathrm{L}} + \sum_{\pi_j \in \pi^{\text{big}}} (1.0 - s\ell) \cdot sp^{\mathrm{L}} \cdot x_{i,j} \\
&\quad + \sum_{\pi_j \in \pi^{\text{LIT}}} (1.0 - s\ell) \cdot sp^{\mathrm{L}} \cdot x_{i,j}.
\end{aligned}$$

Subtracting $u_i - (1.0 - s\ell) \cdot sp^{\mathrm{L}}$ from both sides and then dividing both sides by $(1.0 - s\ell) \cdot sp^{\mathrm{L}}$ yields

$$1.0 \geq \sum_{\pi_j \in \pi^{\text{big}}} x_{i,j} + \sum_{\pi_j \in \pi^{\text{LIT}}} x_{i,j}.$$

Because $\pi = \pi^{\text{big}} \cup \pi^{\text{LIT}}$, we have (3.40).

We have (3.40) for any $\tau_i \in \tau_{\text{act}}^{\text{big}}(t) \cup \tau_{\text{act}}^{\text{LIT}}(t) \cup \tau_{\text{act}}^{\text{glob}}(t) = \tau_{\text{act}}(t)$. This completes the proof of the lemma. □

5.3.5 Implementation

This subsection concerns implementing Ufm-SC-EDF by modifying `SCHED_DEADLINE` to follow USE 1 and USE 2.

5.3.5.1 Data structures

Modifications are made to data structures such that `SCHED_DEADLINE` can more easily recognize when USE 1 or USE 2 are broken.

`dl_rq`. Recall from Section 4.4.1 that each `dl_rq` contains the `struct earliest_dl`. The original members of `earliest_dl` are `curr`, which stores the `deadline` of the `SCHED_DEADLINE` task scheduled on the corresponding CPU (or zero if the scheduled task is not a `SCHED_DEADLINE` task), and `next`, which stores the `deadline` of the pushable (*i.e.*, is unscheduled and has affinity for more than one CPU) task that has the earliest `deadline` on the corresponding `dl_rq`.

Our patch adds two members. The first member is `curr_is_global`, a boolean flag that is set when the task with `deadline` equal to `curr` is in τ^{glob} (*i.e.*, in `SCHED_DEADLINE` terms, when this task has affinity for every CPU in the `span` of the corresponding `dl_rq`'s `root_domain`). Note that the value of `curr_is_global` only has meaning when `curr` is nonzero. The second member is `next_global`, which serves the same function as `next`, but only considers the subset of pushable tasks that are also in τ^{glob}.

`curr_is_global` is relevant to USE 1 because whether or not a task is in τ^{glob} determines its weighted deadline (see Definition 5.8). `curr_is_global` is relevant to USE 2 because USE 2 only considers tasks in τ^{glob}. `next` and `next_global` are used to identify the *next* deadline of a CPU.

▽ **Definition 5.15.** Consider a given CPU and the set of unscheduled `SCHED_DEADLINE` tasks with affinity for said CPU. The *next deadline* of this CPU is the value of the earliest `deadline` belonging to any of said tasks. △

For example, if this given CPU is in π^{big}, then the next deadline is the minimum of the values of `next` for the other CPUs in π^{big} (because any `SCHED_DEADLINE` task on these `dl_rq`s should also have affinity for the given CPU) and the values of `next_global` for the CPUs in π^{LIT} (because only tasks of τ^{glob} on these `dl_rq`s should have affinity for the given CPU).[5] Identifying the next deadline of a given CPU is relevant to maintaining USE 1 (the next deadline corresponds with $d_\ell(t)$ in USE 1).

[5] If $m^{\text{big}} = 1$ or $m^{\text{LIT}} = 1$, then it may not be possible to compute the next deadline of a CPU using only `next` and `next_global`. The problem is that `next` and `next_global` only consider pushable tasks, *i.e.*, tasks with affinity for more than one CPU. For example, suppose $m^{\text{big}} = 1$ and we are interested in the next deadline of a CPU in π^{big}. Suppose the earliest `deadline` of any unscheduled task with affinity for this CPU is already on its `dl_rq`. Because $m^{\text{big}} = 1$, this unscheduled task is not pushable, meaning its `deadline` will not be reflected in this `dl_rq`'s `next` or `next_global` members. We permit this minor bug in our patch because, to our knowledge, architectures with only a single big or LITTLE CPU are rare.

To efficiently update next_global, a new tree named global_dl_tasks_root is added to each dl_rq. The subset of tasks inserted into pushable_dl_tasks_root that are in τ^{glob} are inserted into global_dl_tasks_root. Similarly to how next is the deadline of the leftmost task in pushable_dl_tasks_root, next_global is the deadline of the leftmost task in global_dl_tasks_root. A new rb_nodeglobal_dl_tasks (analogous to pushable_dl_tasks) for insertion onto global_dl_tasks_root is added to each task_struct.

As with the original members of earliest_dl and pushable_dl_tasks_root, the added members curr_is_global and next_global and tree global_dl_tasks_root are updated within enqueue_task_dl() and dequeue_task_dl(). Note that our patch modifies dequeue_task_dl() such that next is set to zero when the last task on pushable_dl_tasks_root is removed. next_global is also set to zero when the last task on global_dl_tasks_root is removed. When computing the next deadline for a CPU, only nonzero next and next_global are considered. In the case that all such values are zero, then, according to our analysis, the next deadline is $t + T_{[1]}$, the deadline of the considered CPU's idle task (see Definition 5.3). In this case, t is the return value of rq_clock() and $T_{[1]}$ is stored in max_dl_period.

We also add an hrtimer named wdl_timer with callback function wdl_check_timer(). The usage of this timer will be detailed in Section 5.3.5.2.

root_domain. The cpudl heap in the root_domain is not suited for a Ufm-SC-EDF implementation because it is oblivious to the asymmetric capacities and affinities present in our platform. As such, we remove it in our patch.

Several new members take the place of the cpudl. We add two cpumask_var_ts, big_online and little_online, which indicate the span of π^{big} and π^{LIT}, respectively. Our implementation assumes that π^{big} consists of the CPUs with a capacity of 1.0 and π^{LIT} consists of the CPUs with a capacity of less than 1.0. min_cpu_capacity stores the minimum capacity of any CPU in little_online. Because our analysis is limited to two distinct speeds, our ACS pessimistically assumes that all CPUs in little_online have capacity equal to min_cpu_capacity. This will be detailed further in Section 5.3.5.3.

max_dl_period stores the value of $T_{[1]}$ for tasks in this root_domain. Note that the analysis in Chapter 3 and in this chapter assumes that the value of $T_{[1]}$ is known *a priori*. Because $T_{[1]}$ is not known *a priori* to SCHED_DEADLINE, max_dl_period is initialized to zero and updated as new tasks are

accepted by the ACS. For our analysis to be applicable, the task that has period equal to $T_{[1]}$ should enter SCHED_DEADLINE prior to any other task. Note that, because $T_{[1]}$ refers to the largest period in the entire task set τ, and not the largest period among the active tasks $\tau_{\text{act}}(t)$, max_dl_period is not decreased if its corresponding task leaves SCHED_DEADLINE.

The values of $d^{\text{big}}\left(\bar{\mathbf{X}}, t\right)$ and $d^{\text{LIT}}\left(\bar{\mathbf{X}}, t\right)$ (where $\bar{\mathbf{X}}$ is the current configuration under our patched SCHED_DEADLINE) are stored in big_deadline and little_deadline, respectively. Storing $d^{\text{big}}\left(\bar{\mathbf{X}}, t\right)$ and $d^{\text{LIT}}\left(\bar{\mathbf{X}}, t\right)$ speeds up the computation of weighted deadlines (see Definition 5.8), which are of relevance to maintaining USE 1. big_deadline is set to the latest earliest_dl.curr value corresponding with any CPU in big_online. A special case is when some CPU in big_online has an curr of zero, *i.e.*, said CPU does not schedule a SCHED_DEADLINE task. In this case, big_deadline is also set to zero. According to the analysis presented in this chapter, the value of $d^{\text{big}}\left(\bar{\mathbf{X}}, t\right)$ should be $t + T_{[1]}$ (*i.e.*, the deadline of an idle task) in this case, where t is the current time. It is not practical to set big_deadline to $t + T_{[1]}$ because this value changes continuously with time. Instead, when a big_deadline value of zero is observed in the code, the value $t + T_{[1]}$ (where t is set to the return value of rq_clock() and $T_{[1]}$ is max_dl_period) is used in place of big_deadline. little_deadline is analogously set based on the CPUs in little_online.

5.3.5.2 Scheduling and Migration Changes

The implementation is mostly unchanged by our patch outside of modifications to find_later_rq() and the addition of two new functions that correspond to USE 1 and USE 2.

find_later_rq(). Recall that when push_dl_task() or select_task_rq_dl() is called on a task being migrated, both functions internally call find_later_rq(). In the original implementation, find_later_rq() references the cpudl heap to identify which CPU has the latest deadline. Because we do not maintain the cpudl heap, when our modified find_later_rq() is called on a task, it must identify the latest CPU by iterating over the CPUs in said task's affinity mask. The latest CPU is the CPU whose dl_rq has the latest earliest_dl.curr value.

Modifying find_later_rq() to iterate over the CPUs in a task's affinity mask prevents push_dl_task() and select_task_rq_dl() from failing to migrate a task due to find_later_rq() returning a CPU that said task does not have affinity for. Recall that pull_dl_task() does not fail

to migrate tasks in this manner because `pull_dl_task()` does not reference an affinity-oblivious data structure such as the `cpudl` heap. Because CPUs pull the earliest unscheduled tasks from other `dl_rq`s and tasks are now correctly pushed to the latest CPUs they have affinity for, `SCHED_DEADLINE`, under these modifications implements Weak-APA-EDF.[6] The addition of two new functions extends Weak-APA-EDF to Ufm-SC-EDF. Both functions are callback functions that are invoked by waking the `stop` tasks of CPUs that schedule tasks in τ^{glob} (*i.e.*, a task with affinity for every CPU in the `root_domain`'s `span`).

USE 1 and `set_next_task_dl()`. Because `SCHED_DEADLINE` already implements Weak-APA-EDF, any unscheduled task τ_ℓ must have a later deadline $d_\ell(t)$ than any task τ_e with deadline $d_e(t)$ that is scheduled on a CPU π_j that τ_ℓ has affinity for. Recall from Definition 5.8 that only a scheduled task in $\tau_e \in \tau^{\text{glob}}$ can potentially have $\bar{d}_e\big(\bar{\mathbf{X}}, t\big) \neq d_e(t)$. Thus, we only need to check for violations of USE 1 whenever a task $\tau_e \in \tau^{\text{glob}}$ is newly scheduled on a CPU π_j or when the weighted deadline $\bar{d}_e\big(\bar{\mathbf{X}}, t\big)$ of task τ_e scheduled on a CPU π_j changes.

We first address scenarios when a task $\tau_e \in \tau^{\text{glob}}$ is newly scheduled on a CPU π_j We modify `set_next_task_dl()` to call a new function `check_wdl_preempt()` that checks for violations of USE 1. $\bar{d}_e\big(\bar{\mathbf{X}}, t\big)$ is computed by referencing task τ_e's `deadline` (*i.e.*, $d_e(t)$) and the added fields `min_cpu_capacity` (*i.e.*, sp^{L}), `big_deadline` (*i.e.*, $d^{\text{big}}\big(\bar{\mathbf{X}}, t\big)$), and `little_deadline` (*i.e.*, $d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big)$). These values are plugged into the formula presented in Definition 5.8. The earliest deadline $d_\ell(t)$ of any unscheduled task τ_ℓ with affinity for CPU π_j is π_j's next deadline (see Definition 5.15), which, as discussed in Section 5.3.5.1, is computed using fields `next` and `next_global`.

If USE 1 is found to be violated in `check_wdl_preempt()` for a task $\tau_e \in \tau^{\text{glob}}$ scheduled on a CPU π_j, `set_next_task_dl()` calls `stop_one_cpu_nowait()` on CPU π_j with callback function `push_wdl_stop()`, which is added by our patch. When `push_wdl_stop()` is executed by the `stop` task, the target CPU is the latest LITTLE CPU if $\pi_j \in \pi^{\text{big}}$ and the target CPU is the latest big CPU if $\pi_j \in \pi^{\text{LIT}}$. `push_wdl_stop()` acquires the `rq` locks of CPU π_j and the target CPU, after which it rechecks that $\tau_e \in \tau^{\text{glob}}$ is still the highest-priority `SCHED_DEADLINE` task on π_j and that the highest-priority `SCHED_DEADLINE` task on the target CPU still has a later `deadline` than $d_e(t)$. If this recheck passes, task τ_e is pushed from CPU π_j to the target CPU. `push_wdl_stop()` then returns, allowing CPU

[6] We claim that `SCHED_DEADLINE` with these changes implements Weak-APA-EDF without formal proof. It is impractical to formally reason about `SCHED_DEADLINE` given the size of the Linux scheduler. Our justification that `SCHED_DEADLINE` implements Weak-APA-EDF is that the changes made in this patch mitigate the affinity-related issues discussed in Section 4.4.5. These issues are the cause of the counterexample in Example 4.10.

π_j's `stop` task to suspend. When π_j reschedules due to the `stop` task suspending, it pulls and schedules the task τ_ℓ that was involved in the violation of USE 1.

Changes in weighted deadlines. It remains to discuss when USE 1 is violated due to the weighted deadline $\bar{d}_e\big(\bar{\mathbf{X}}, t\big)$ of task τ_e, which is scheduled on CPU π_j, changing. By Definition 5.8, this can only occur if either $d_e(t)$ (*i.e.*, the `deadline` of task τ_e), $d^{\text{big}}\big(\bar{\mathbf{X}}, t\big)$ (*i.e.*, `big_deadline`), or $d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big)$ (*i.e.*, `little_deadline`) changes.

Instances of $d_e(t)$ changing are dealt with implicitly by eliminating throttle bypassing. In `SCHED_DEADLINE`, the `deadline` of a task can only change upon wakeup or replenishment (be aware that this assumes no priority inheritance). If the task was woken, then it was necessarily unscheduled while suspended. If the task was replenished, then without throttle bypassing, it was necessarily unscheduled while throttled. Thus, any task that changed its `deadline` will first need to be rescheduled, upon which the task will encounter the aforementioned checks in `set_next_task_dl()`.

Instances of $d^{\text{big}}\big(\bar{\mathbf{X}}, t\big)$ or $d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big)$ changing are dealt with by calling `resched_curr()` on certain CPUs. If $d^{\text{big}}\big(\bar{\mathbf{X}}, t\big)$ is set to a new value in `enqueue_task_dl()` or `dequeue_task_dl()`, then the weighted deadlines of tasks in τ^{glob} that are scheduled on CPUs in π^{LIT} may have changed. `enqueue_task_dl()` or `dequeue_task_dl()` then calls `resched_curr()` on these CPUs in π^{LIT}. These CPUs are those for which `earliest_dl.curr` is nonzero and `earliest_dl.curr_is_global` is set. Likewise, if $d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big)$ is set to a new value, `enqueue_task_dl()` or `dequeue_task_dl()` calls `resched_curr()` on such CPUs in π^{big}. CPUs that have `resched_curr()` called on them in this manner will check for USE 1 violations using the updated $d^{\text{big}}\big(\bar{\mathbf{X}}, t\big)$ and $d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big)$ values in `set_next_task_dl()`.

Non-`SCHED_DEADLINE` tasks. There is an edge case when a CPU is not executing a `SCHED_DEADLINE` task. If a CPU in π^{big} does not schedule a `SCHED_DEADLINE` task, then $d^{\text{big}}\big(\bar{\mathbf{X}}, t\big) = t + T_{[1]}$ (by Definition 5.3, 5.6, and 5.7 and Corollary 5.3). Likewise, if a CPU in π^{LIT} does not schedule a `SCHED_DEADLINE` task, then $d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big) = t + T_{[1]}$. Because $d^{\text{big}}\big(\bar{\mathbf{X}}, t\big)$ or $d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big)$ changes continuously with time while a CPU executes a non-`SCHED_DEADLINE` task, it no longer becomes practical to call `resched_curr()` on the relevant CPUs upon every change in $d^{\text{big}}\big(\bar{\mathbf{X}}, t\big)$ or $d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big)$. Instead, `resched_curr()` is called on these CPUs only when `big_deadline` or `little_deadline` is set to zero (recall that this signifies that $d^{\text{big}}\big(\bar{\mathbf{X}}, t\big)$ or $d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big)$ is equal to $t + T_{[1]}$).

When `resched_curr()` is called on a CPU due to `big_deadline` or `little_deadline` being set to zero, `set_next_task_dl()` will perform the aforementioned checks if USE 1 is violated. If it is found that USE 1 is violated (while using the sum of `rq_clock()` and `max_dl_period` in place of `big_deadline` or `little_deadline`), then the relevant tasks are migrated as described above. Otherwise, USE 1 is not violated, and the issue becomes that USE 1 may become violated at some future time t' due to $t + T_{[1]}$ (and hence, $d^{\text{big}}\big(\bar{\mathbf{X}}, t\big)$ or $d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big)$) increasing with time. To see this, observe from Definition 5.8 that the weighted deadline of a task is non-decreasing with $d^{\text{big}}\big(\bar{\mathbf{X}}, t\big)$ and $d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big)$, thus, a continuous increase in $d^{\text{big}}\big(\bar{\mathbf{X}}, t\big)$ or $d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big)$ can lead to a violation of USE 1 by increasing a scheduled task's weighted deadline.

The time instant t' is computed in `set_next_task_dl()`, which arms `hrtimer` `wdl_timer` to fire at time t'. `wdl_check_timer()`, `wdl_timer`'s callback function, triggers a reschedule on this CPU, which will result in a call to `set_next_task_dl()` that will observe the violation of USE 1 at time t'.

The computation of time t' in `set_next_task_dl()` makes three assumptions. Note that these assumptions need not be true; the violation of any of these assumptions only means that the time t' must be recomputed and `wdl_timer` must be armed to fire at the new time. The computation assumes that from the time `set_next_task_dl()` is called to the time t':

- the task $\tau_e \in \tau^{\text{glob}}$ being scheduled in `set_next_task_dl()` remains scheduled on the same CPU,
- this task τ_e has constant `deadline` $d_e(t)$,
- and the next deadline of task τ_e's CPU is constant.

We now justify these three assumptions.

It is assumed that τ_e remains scheduled until time t' because USE 1 only considers $\bar{d}_e\big(\bar{\mathbf{X}}, t\big)$ while task τ_e is scheduled. If τ_e is later rescheduled, then we can defer checking for violations of USE 1 to the call of `set_next_task_dl()` that corresponds with that rescheduling. If necessary, that call to `set_next_task_dl()` will recompute t'.

It is assumed that $d_e(t)$ is constant until time t' because, due to eliminating throttle bypassing, τ_e must have been unscheduled for $d_e(t)$ to have changed. As with the prior assumption, `set_next_task_dl()` will recompute t' if τ_e is rescheduled.

It is not safe to assume that the next deadline of task τ_e's CPU is constant until time t'. If a task replaces task τ_ℓ as the unscheduled task with the earliest `deadline` with affinity for this CPU, the implementation

needs to explicitly call `resched_curr()` to trigger a call to `set_next_task_dl()` such that t' will be recomputed. We will address how the patch does this after discussing how t' is computed given the above assumptions.

Suppose we have a CPU in π^{LIT} that schedules a non-`SCHED_DEADLINE` task and a CPU in π^{big} that schedules a task $\tau_e \in \tau^{\text{glob}}$. Task $\tau_\ell \in \bar{\tau}_{\text{rdy}}(t)$ is the unscheduled task whose `deadline` corresponds with the next deadline of task τ_ℓ's CPU. USE 1 is violated immediately after the time instant t' such that $\bar{d}_e\big(\bar{\mathbf{X}}, t'\big) = d_\ell(t')$. This time instant is

$$t' = \frac{d_\ell(t) - \big(1.0 - sp^{\text{L}}\big) \cdot d_e(t)}{sp^{\text{L}}} - T_{[1]}, \tag{5.22}$$

This is because

$$\begin{aligned}
\bar{d}_e\big(\bar{\mathbf{X}}, t'\big) &= \big\{\text{Definition 5.8 and } \tau_e \in \tau^{\text{glob}} \text{ is scheduled on a big CPU}\big\} \\
&\quad \big(1.0 - sp^{\text{L}}\big) \cdot d_e\big(t'\big) + sp^{\text{L}} \cdot d^{\text{LIT}}\big(\bar{\mathbf{X}}, t'\big) \\
&= \big\{\text{Assumed constant } d_e(t) \text{ from current time to } t'\big\} \\
&\quad \big(1.0 - sp^{\text{L}}\big) \cdot d_e(t) + sp^{\text{L}} \cdot d^{\text{LIT}}\big(\bar{\mathbf{X}}, t'\big) \\
&= \big\{d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big) = t + T_{[1]} \text{ while a CPU in } \pi^{\text{LIT}} \text{ schedules a non-}\texttt{SCHED_DEADLINE}\text{ task}\big\} \\
&\quad \big(1.0 - sp^{\text{L}}\big) \cdot d_e(t) + sp^{\text{L}} \cdot \big(t' + T_{[1]}\big) \\
&= \{\text{Equation (5.22)}\} \\
&\quad \big(1.0 - sp^{\text{L}}\big) \cdot d_e(t) + sp^{\text{L}} \cdot \left(\frac{d_\ell(t) - \big(1.0 - sp^{\text{L}}\big) \cdot d_e(t)}{sp^{\text{L}}} - T_{[1]} + T_{[1]}\right) \\
&= \big(1.0 - sp^{\text{L}}\big) \cdot d_e(t) + sp^{\text{L}} \cdot \frac{d_\ell(t) - \big(1.0 - sp^{\text{L}}\big) \cdot d_e(t)}{sp^{\text{L}}} \\
&= \big(1.0 - sp^{\text{L}}\big) \cdot d_e(t) + d_\ell(t) - \big(1.0 - sp^{\text{L}}\big) \cdot d_e(t) \\
&= d_\ell(t) \\
&= \big\{\text{Assumed next deadline } d_\ell(t) \text{ is constant until } t'\big\} \\
&\quad d_\ell\big(t'\big).
\end{aligned}$$

Likewise, if we have a CPU in π^{big} that schedules a non-`SCHED_DEADLINE` task, a CPU in π^{LIT} that schedules a task $\tau_e \in \tau^{\text{glob}}$, and task τ_e's CPU's next deadline is $d_\ell(t)$ for some task $\tau_\ell \in \bar{\tau}_{\text{rdy}}(t)$, then we

have

$$t' = sp^{\mathrm{L}} \cdot d_\ell(t) + \big(1.0 - sp^{\mathrm{L}}\big) \cdot d_e(t) - T_{[1]}. \tag{5.23}$$

This is because

$$\begin{aligned}
\bar{d}_e\big(\bar{\mathbf{X}}, t'\big) &= \big\{\text{Definition 5.8 and } \tau_e \in \tau^{\text{glob}} \text{ is scheduled on a LITTLE CPU}\big\} \\
&\quad \frac{1.0}{sp^{\mathrm{L}}} \cdot d^{\text{big}}\big(\bar{\mathbf{X}}, t'\big) - \frac{1.0 - sp^{\mathrm{L}}}{sp^{\mathrm{L}}} \cdot d_e\big(t'\big) \\
&= \big\{\text{Assumed constant } d_e(t) \text{ from current time to } t'\big\} \\
&\quad \frac{1.0}{sp^{\mathrm{L}}} \cdot d^{\text{big}}\big(\bar{\mathbf{X}}, t'\big) - \frac{1.0 - sp^{\mathrm{L}}}{sp^{\mathrm{L}}} \cdot d_e(t) \\
&= \big\{d^{\text{big}}\big(\bar{\mathbf{X}}, t\big) = t + T_{[1]} \text{ while a CPU in } \pi^{\text{big}} \text{ schedules a non-}\texttt{SCHED_DEADLINE}\text{ task}\big\} \\
&\quad \frac{1.0}{sp^{\mathrm{L}}} \cdot \big(t' + T_{[1]}\big) - \frac{1.0 - sp^{\mathrm{L}}}{sp^{\mathrm{L}}} \cdot d_e(t) \\
&= \{\text{Equation (5.23)}\} \\
&\quad \frac{1.0}{sp^{\mathrm{L}}} \cdot \big(sp^{\mathrm{L}} \cdot d_\ell(t) + \big(1.0 - sp^{\mathrm{L}}\big) \cdot d_e(t) - T_{[1]} + T_{[1]}\big) - \frac{1.0 - sp^{\mathrm{L}}}{sp^{\mathrm{L}}} \cdot d_e(t) \\
&= \frac{1.0}{sp^{\mathrm{L}}} \cdot \big(sp^{\mathrm{L}} \cdot d_\ell(t) + \big(1.0 - sp^{\mathrm{L}}\big) \cdot d_e(t)\big) - \frac{1.0 - sp^{\mathrm{L}}}{sp^{\mathrm{L}}} \cdot d_e(t) \\
&= d_\ell(t) + \frac{\big(1.0 - sp^{\mathrm{L}}\big)}{sp^{\mathrm{L}}} \cdot d_e(t) - \frac{1.0 - sp^{\mathrm{L}}}{sp^{\mathrm{L}}} \cdot d_e(t) \\
&= d_\ell(t) \\
&= \big\{\text{Assumed next deadline } d_\ell(t) \text{ is constant until } t'\big\} \\
&\quad d_\ell\big(t'\big).
\end{aligned}$$

Changes in the next deadline. The computation of time t' and the arming of `wdl_timer` assume that the next deadline is constant until time t'. The code must detect when this assumption is broken in order to know when to recompute t' and rearm `wdl_timer`. Recall from the discussion in Section 5.3.5.1 that the next deadline is computed as the minimum of certain `earliest_dl.next` and `earliest_dl.next_global` values, which are updated within functions `enqueue_task_dl()` and `dequeue_task_dl()`. We modify `enqueue_task_dl()` and `dequeue_task_dl()` such that, if any `next` or `next_global` is set to a lesser value while `big_deadline` or `little_deadline` is zero (recall

that this signifies that $d^{\text{big}}\big(\bar{\mathbf{X}}, t\big) = t + T_{[1]}$ or $d^{\text{LIT}}\big(\bar{\mathbf{X}}, t\big) = t + T_{[1]}$, which is the cause for needing to compute time t' and arm `wdl_timer`), then `resched_curr()` is called on the CPUs that schedule tasks in τ^{glob} (*i.e.*, CPUs with `curr_is_global` set to `true`). After rescheduling, `set_next_task_dl()` will recompute time t' and arm `wdl_timer` if necessary.

USE 2. USE 2 can only be violated when a task in τ^{glob} is newly scheduled on a CPU. As with USE 1, we check `curr_is_global` within `set_next_task_dl()` to determine if checking for a violation is necessary, and, if a violation of USE 2 is discovered, execute a callback function to perform a migration. The function for checking USE 2 is `check_global_order()`. The callback function for USE 2 is `swap_global_stop()`, the purpose of which is to swap the CPUs of a high-priority task $\tau_{i_1} \in \tau^{\text{glob}}$ that is scheduled on a CPU $\pi_{j_1} \in \pi^{\text{LIT}}$ and a lower-priority task $\tau_{i_2} \in \tau^{\text{glob}}$ that is scheduled on a CPU $\pi_{j_2} \in \pi^{\text{big}}$.

How `check_global_order()` checks for violations of USE 2 depends on whether the calling CPU belongs to π^{big} or π^{LIT}. If the calling CPU belongs to π^{big}, then it corresponds with CPU π_{j_2} and its scheduled task corresponds with task τ_{i_2}. A violation of USE 2 has occurred if there is a CPU $\pi_{j_1} \in \pi^{\text{LIT}}$ that schedules a task $\tau_{i_1} \in \tau^{\text{glob}}$ such that task τ_{i_1} has an earlier `deadline` than task τ_{i_2}, *i.e.*, $d_{i_1}(t) < d_{i_2}(t)$. The code determines if this has occurred by comparing the `deadline` of τ_{i_2} against `earliest_dl.curr` for each CPU $\pi_{j_1} \in \pi^{\text{big}}$ where `earliest_dl.curr_is_global` is set. Likewise, if the calling CPU belongs to π^{LIT}, then it corresponds with CPU π_{j_1} and its scheduled task corresponds with task τ_{i_1}. A violation of USE 2 has occurred if there is a CPU $\pi_{j_2} \in \pi^{\text{big}}$ that schedules a task $\tau_{i_2} \in \tau^{\text{glob}}$ such that $d_{i_1}(t) < d_{i_2}(t)$. The code determines if this has occurred by comparing the `deadline` of τ_{i_1} against `earliest_dl.curr` for each CPU $\pi_{j_2} \in \pi^{\text{LIT}}$ where `earliest_dl.curr_is_global` is set.

If a violation is discovered, `swap_global_stop()` is called on CPUs π_{j_1} and π_{j_2} using `stop_two_cpus()`.[7] `swap_global_stop()` first acquires the locks of the calling and target CPUs. `swap_global_stop()` then rechecks that the highest-priority `SCHED_DEADLINE` tasks on both CPUs are still tasks in τ^{glob} and that the task belonging to the LITTLE CPU still has an earlier `deadline` than the `deadline` of the task belonging to the big CPU. If the recheck passes, `swap_global_stop()` migrates the tasks such that they swap CPUs.

[7]More specifically, `swap_global_stop()` is called from a non-blocking version of `stop_two_cpus()`, which is added in our patch. A non-blocking version is necessary to avoid nested calls of `__schedule()`. Note that `stop_two_cpus()` is being called from `set_next_task_dl()`, which is called by `__schedule()`. If `stop_two_cpus()` were to block, it would call `__schedule()` to select a new task to run.

Note that it is redundant to check for violations of both USE 1 and USE 2 within the same instance of `set_next_task_dl()` because a violation in either will result in the scheduled task being migrated away. Thus, `set_next_task_dl()` will only call `check_wdl_preempt()` to check USE 1 if `check_global_order()` finds that USE 2 is upheld. This ordering of `check_wdl_preempt()` and `check_global_order()` in our patch is arbitrary.

5.3.5.3 ACS

Our patch must modify the ACS to enforce (5.14)-(5.18), which were derived in Section 5.3.4. We now discuss how the terms in conditions (5.14)-(5.18) are stored. Some terms are already maintained by the original code, which we briefly review. Consider a given `root_domain`. Recall from Section 4.4.1 that `dl_bw.bw` stores the fraction $\frac{\texttt{sched\_rt\_runtime\_us}}{\texttt{sched\_rt\_period\_us}}$. As discussed in Section 5.3.4, we choose to interpret this fraction as $1.0 - s\ell$. Member `dl_bw.total_bw` stores the total of the `dl_bw`s of the `SCHED_DEADLINE` tasks executing on the CPUs belonging to this `root_domain`, *i.e.*, `total_bw` corresponds with $\sum_{\tau_i \in \tau_{\text{act}}(t)} u_i$.

Some terms in (5.14)-(5.18) added by our patch have already been discussed. The number of CPUs in `big_online` corresponds with m^{big} and the number of CPUs in `little_online` corresponds with m^{LIT}. `min_cpu_capacity` corresponds with sp^{L}.

We add members `dl_bw.total_big_bw`, which corresponds with $\sum_{\tau_i \in \tau_{\text{act}}^{\text{big}}(t)} u_i$; `dl_bw.total_little_bw`, which corresponds with $\sum_{\tau_i \in \tau_{\text{act}}^{\text{LIT}}(t)} u_i$; and `dl_bw.total_global_b_bw`, which corresponds with $\sum_{\tau_i \in \tau_{\text{act}}^{\text{glob}}(t)} u_i^{\text{big}}$. Note that the additional 'b' in `total_global_b_bw` indicates that the sum of u_i^{big}'s, not u_i's, is being stored. As with `total_bw`, these three members are updated whenever a task enters or leaves `SCHED_DEADLINE`.

Thus, every term in (5.14)-(5.18) corresponds to some variable in `SCHED_DEADLINE`. We modify `__dl_overflow()` to verify that (5.14)-(5.18) would not be violated upon enacting a change in a task's bandwidth. Note that it is necessary to add an additional argument to `__dl_overflow()` that indicates the affinity of the task being considered. For example, only tasks with affinity for the LITTLE CPUs must satisfy (5.15).

For the same reasons as our patch targeting IDENTICAL/SEMI-PARTITIONED systems (recall Section 5.2.4), this patch forbids `SCHED_DEADLINE` tasks from changing their affinities. Tasks are categorized

into $\tau_{\text{act}}^{\text{big}}(t)$, $\tau_{\text{act}}^{\text{glob}}(t)$, or $\tau_{\text{act}}^{\text{LIT}}(t)$ based on their affinities upon entering `SCHED_DEADLINE`. To change a task's affinity, said task must first exit `SCHED_DEADLINE`.

Furthermore, we forbid the addition or removal of CPUs to any `cpuset` that contains `SCHED_DEADLINE` tasks. There are two reasons for forbidding this. The first reason is that modifying a `cpuset` can potentially change the `span` of a `root_domain`, which resets the affinity of every task corresponding with that `root_domain` to said `root_domain`'s `span`. The second reason is that a new CPU added to a `root_domain`'s `span` may have capacity less than `min_cpu_capacity`, which would set `min_cpu_capacity` to this new lesser capacity. Recall that, because our model only permits two distinct capacities, we pessimistically assume that all LITTLE CPUs have capacity `min_cpu_capacity`. Tasks that were already accepted by the ACS under the old `min_cpu_capacity` (*i.e.*, sp^{L}) value may no longer satisfy (5.14)-(5.18) with the new value. Modifying `min_cpu_capacity` would also require that u_i^{big} be recomputed for every task in $\tau_{\text{act}}^{\text{glob}}(t)$ (see Definition 5.14).

Modifying the fraction $\frac{\texttt{sched\_rt\_runtime\_us}}{\texttt{sched\_rt\_period\_us}}$ is also forbidden because this fraction corresponds with $1.0 - s\ell$. Thus, decreasing $\frac{\texttt{sched\_rt\_runtime\_us}}{\texttt{sched\_rt\_period\_us}}$ can cause tasks that were previously accepted by the ACS to no longer satisfy (5.14)-(5.18). Similarly to `min_cpu_capacity`, changing this fraction would also require that u_i^{big} be recomputed for every task in $\tau_{\text{act}}^{\text{glob}}(t)$ (see Definition 5.14).

5.3.6 Evaluation

We compared the overheads and tardiness under our patched kernel against those of the original implementation on the ODROID-XU4. As with the evaluation discussed in Section 5.2.6, `rt-app` (rt-app) was used for workload generation and `ftrace` was used for recording timestamps. We modified `taskgen` to generate bandwidths and affinities satisfying (5.14)-(5.18), the conditions checked by our modified ACS. Task parameter generation under our modified `taskgen` is detailed in the following paragraphs. Ten task systems were generated for each of $n \in \{20, 40\}$. Timestamps were collected for each task system for 10 minutes.

Task parameter generation. Recall from Section 5.3.1 that, on our considered platform, we have $m^{\text{big}} = m^{\text{LIT}} = 4$. The capacity of each LITTLE CPU is $sp^{\text{L}} = \frac{377}{1024} \approx 0.368$. The value of $1.0 - s\ell = \frac{\texttt{sched\_rt\_runtime\_us}}{\texttt{sched\_rt\_period\_us}}$ was kept at its default of 95%. We generated task systems satisfying (5.14)-(5.18) with $n \in \{20, 40\}$ using the following procedure. Note that, for each task system, all tasks both simultaneously

enter and simultaneously leave `SCHED_DEADLINE`, *i.e.*, for all time instants t while a task system τ is being evaluated, we have $\tau_{\text{act}}(t) = \tau$. Our generation procedure initially generates a system of only global tasks (Step 1 through Step 3), after which certain tasks are partitioned out into τ^{LIT} (Step 4) and τ^{big} (Step 5).

Step 1: Use Randfixedsum (Stafford, 2024), the algorithm used internally by `taskgen` (Emberson et al., 2010; Lelli, 2014), to uniformly generate a set of n utilizations $u_i \in [0, 1.0]$ that sum to $\sum_{\tau_i \in \tau} u_i = m^{\text{big}} + sp^{\text{L}} \cdot m^{\text{LIT}}$.

Step 2: Scale each utilization such that $u_i \leftarrow (1.0 - s\ell) \cdot u_i$.

Each utilization u_i is now in $[(1.0 - s\ell) \cdot 0, (1.0 - s\ell) \cdot 1.0] = [0, 1.0 - s\ell]$, which satisfies (5.14). The sum of all utilizations is now $\sum_{\tau_i \in \tau} u_i = (1.0 - s\ell) \cdot \left(m^{\text{big}} + sp^{\text{L}} \cdot m^{\text{LIT}}\right)$, which satisfies (5.18).

Step 3: Compute u_i^{big} for each utilization u_i. If $\sum_{\tau_i \in \tau} u_i^{\text{big}} > (1.0 - s\ell) \cdot m^{\text{big}}$, then restart from Step 1.

Note that it is unlikely that the procedure returns to Step 1 at the end of Step 3, which we argue informally. The average utilization is $\frac{\sum_{\tau_i \in \tau} u_i}{n} \leq \frac{\sum_{\tau_i \in \tau} u_i}{20} = \frac{(1.0 - s\ell) \cdot \left(m^{\text{big}} + sp^{\text{L}} \cdot m^{\text{LIT}}\right)}{20} \approx \frac{0.95 \cdot (4 + 0.368 \cdot 4)}{20} \approx 0.260$. Thus, the average utilization is less than $(1.0 - s\ell) \cdot sp^{\text{L}} \approx 0.95 \cdot 0.368 \approx 0.350$. Because the average utilization is less than $(1.0 - s\ell) \cdot sp^{\text{L}}$, by Definition 5.14, an average utilization u_i has a corresponding $u_i^{\text{big}} = 0$. Thus, it is unlikely that $\sum_{\tau_i \in \tau} u_i^{\text{big}} > (1.0 - s\ell) \cdot m^{\text{big}} = 0.95 \cdot 4 = 3.8$.

Passing Step 3 satisfies (5.16) for our initial task system, which consists of only global tasks. Note that Step 5, which will move tasks from τ^{glob} into τ^{big}, must do so without violating (5.16).

Step 4: Sort the tasks of $\tau = \tau^{\text{glob}}$ in order of non-decreasing utilization. Move tasks of τ^{glob} into τ^{LIT} until either of constraints (5.15) or (5.17) would be violated or until $\tau^{\text{LIT}} = \tau$.

The tasks are sorted in non-decreasing order because this maximizes the number of tasks that can be placed in τ^{LIT} without violating (5.15) or (5.17). We assume that a system designer would, to save power, prefer that as many tasks as possible execute exclusively on the LITTLE CPUs.

Step 5: Move the remaining tasks of τ^{glob} into τ^{big} until (5.16) would be violated or until $\tau^{\text{big}} = \tau \setminus \tau^{\text{LIT}}$.

Note that these remaining tasks of τ^{glob} are still sorted in order of non-decreasing utilization. As when moving tasks from τ^{glob} to τ^{LIT} in Step 4, this order maximizes the number of tasks moved from τ^{glob} to τ^{big}. While there is less benefit in maximizing the number of tasks that execute exclusively on big CPUs than on LITTLE CPUs (exclusively scheduling tasks on big CPUs does not save power), the rate

of L2 cache misses should decrease as more tasks are moved into τ^{big} (recall from Figure 5.8 that the big CPUs share an L2 cache).

Step 6: Sample a period T_i for each utilization u_i from the log-uniform distribution (as proposed by Emberson et al. (2010)) ranging from 10 ms to 1 s. Compute C_i as $T_i \cdot u_i$.

As discussed by Emberson et al. (2010), periods are sampled from a log-uniform distribution because sampling from a purely uniform distribution rarely generates periods at the lesser extreme. For example, uniformly sampling the range from 10 ms to 1 s has a less than 10% chance of generating a period of less than 100 ms. The log-uniform distribution biases samples such that periods that are on the order of 10 ms have a higher chance of being generated.

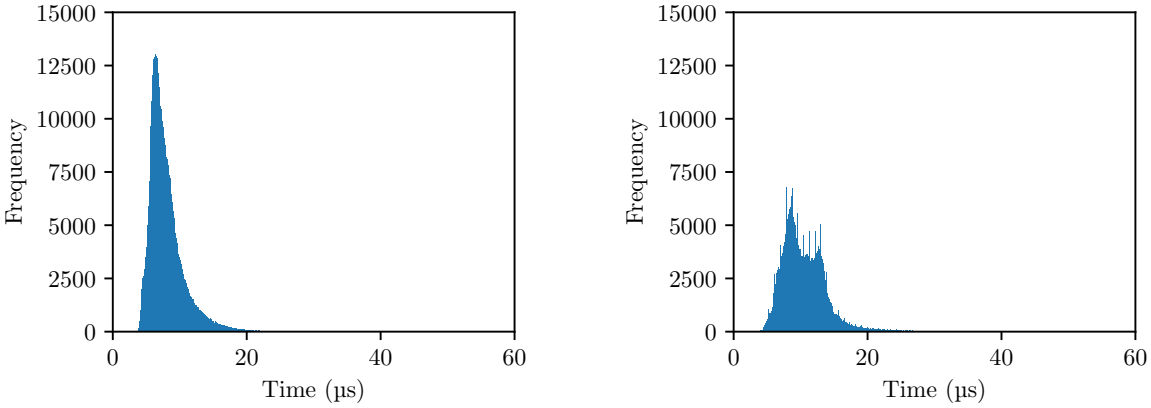
Note that the above task generation procedure results in a low number of tasks in τ^{glob}, especially for systems where $n = 40$. Similarly to how bin-packing efficiency increases as the average weight of items decreases, at higher task counts, the average task's bandwidth becomes low enough that almost all tasks can be placed into τ^{big} or τ^{LIT}. Systems generated with $n = 20$ tend to have around three tasks in τ^{glob}. All evaluated systems with $n = 40$ had *at most* one task in τ^{glob}.

The low number of tasks in τ^{glob} influences the overhead measurements. As discussed in Section 5.3.5.2, our newly added functions `push_wdl_stop()` and `swap_global_stop()` can only be called when tasks of τ^{glob} are scheduled. We argue that a low number of tasks in τ^{glob} is realistic. A system designer has incentive to pack as many tasks into τ^{big} and τ^{LIT} as possible in order to reduce migration and cache miss overheads.

Enqueueing and dequeueing. A modification made by our patch is the removal of the `cpudl` and the addition of members `curr_is_global` and `next_global` to `earliest_dl` and of members `big_deadline` and `little_deadline` to `root_domain`. These data structures are all maintained on calls to `enqueue_task_dl()` and `dequeue_task_dl()`. We measured the duration of these function calls in the original kernel and in our patched kernel.[8]

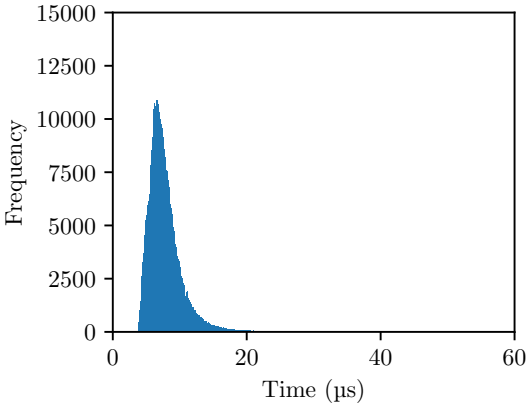
Histograms illustrating the distribution of these call durations are presented in Figures 5.21 and 5.22. For both $n = 20$ and $n = 40$, in both the original kernel and our patched kernel, a majority of call durations fall under 20 µs for `enqueue_task_dl()` and 30 µs for `dequeue_task_dl()`. For `enqueue_task_`

[8]Note that not all of `enqueue_task_dl()` and `dequeue_task_dl()` are included in these measurements. For example, code pertaining to GRUB and CBS logic (see Listing 4.21) is excluded because it is unaffected by our patch.

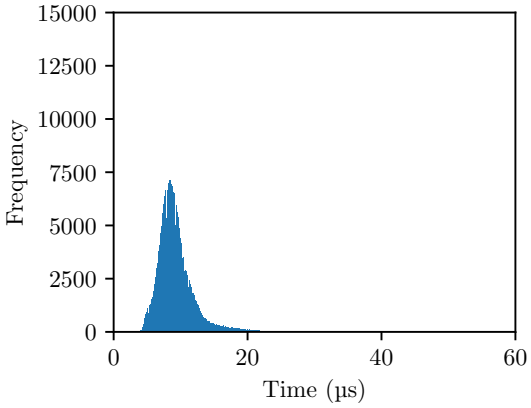


(a) Original ($n = 20$).

(b) Patched ($n = 20$).

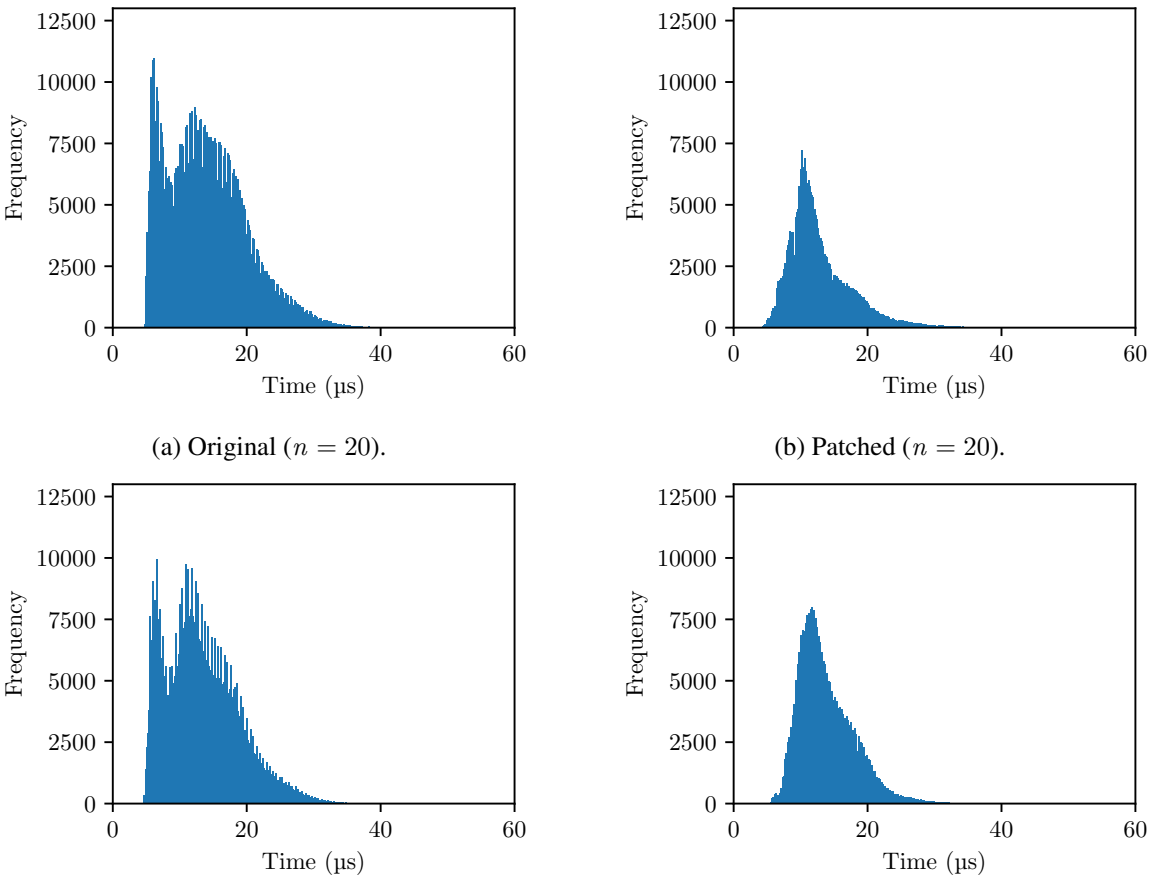


(c) Original ($n = 40$).



(d) Patched ($n = 40$).

Figure 5.21: `enqueue_task_dl()` overhead.



(a) Original ($n = 20$).

(b) Patched ($n = 20$).

(c) Original ($n = 40$).

(d) Patched ($n = 40$).

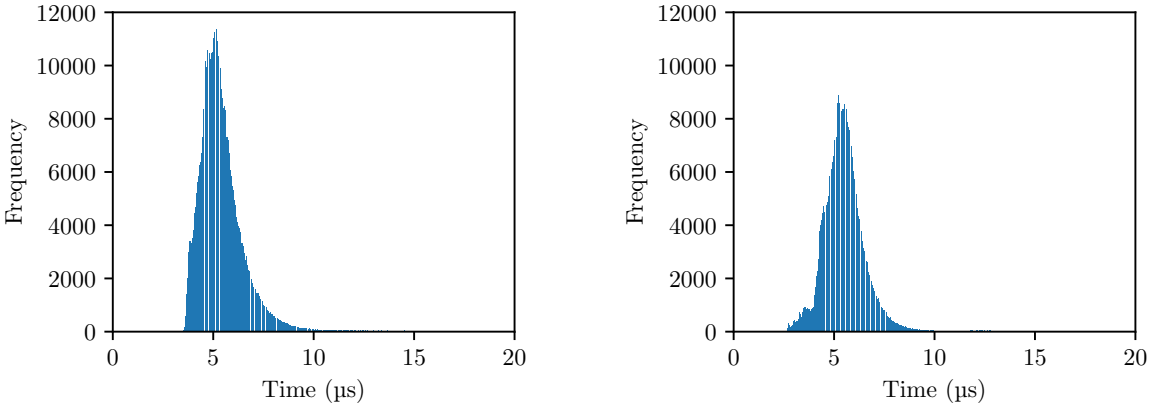
Figure 5.22: `dequeue_task_dl()` overhead.

dl(), worst-case measurements were roughly 50 µs for the original kernel and 60 µs for our patched kernel (this was consistent for both $n = 20$ and $n = 40$). For dequeue_task_dl(), worst-case measurements were roughly 60 µs for both kernels and n values. This shows that the overhead involved in maintaining the cpudl in the original kernel is roughly equivalent to the overhead involved in maintaining big_deadline and little_deadline in our patched kernel.

Note that this equivalence in overheads is unlikely to scale to higher CPU counts (within the same root_domain), as the cpudl is organized as a heap, while big_deadline and little_deadline are updated by iterating over the CPUs. In our patched kernel, this could be mitigated by using two heaps, one for the CPUs in π^{big} and another for those in π^{LIT}, to maintain big_deadline and little_deadline. We chose not to implement this because there are only four CPUs in both π^{big} and in π^{LIT} on the ODROID-XU4, so maintaining two heaps of only four elements each would be wasteful. We also argue that there is little incentive to run SCHED_DEADLINE with a single root_domain containing a large number of CPUs. The fraction of CPU capacity lost by partitioning the CPUs into several root_domains quickly decreases with the size of the root_domains.

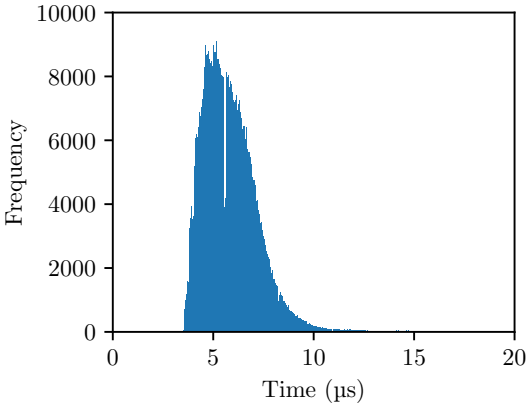
find_later_rq(). Because our patched kernel removes the cpudl, find_later_rq() is required to iterate over the CPUs in the task being migrated's affinity mask to discover which of these CPUs schedules the task with the latest deadline (or schedules a non-SCHED_DEADLINE task). Histograms of function call durations are presented in Figure 5.23. A majority of find_later_rq() calls take around 10 µs for both kernels and n values. Both kernels had worst-case find_later_rq() durations of roughly 40 µs. Our patch does not significantly impact the overhead of calling find_later_rq(), though, as with enqueue_task_dl() and dequeue_task_dl(), this is unlikely to scale with larger CPU counts.

check_wdl_preempt() and push_wdl_stop(). Our patched kernel introduces overheads for checking for violations of USE 1 and USE 2 and migrations that occur in response to such violations. Because these overheads are unique to our patched kernel, we do not compare against the original kernel. Violations of USE 1 are checked in check_wdl_preempt(). Durations of calls to check_wdl_preempt() are presented in Figure 5.24. A majority of calls complete within 10 µs, and the worst-case call observed during measurement took roughly 40 µs. This overhead is within the same order of magnitude as that of calling enqueue_task_dl(), dequeue_task_dl(), and find_later_rq(). As such, the addition of these checks for USE 1 are not a significant source of overhead.

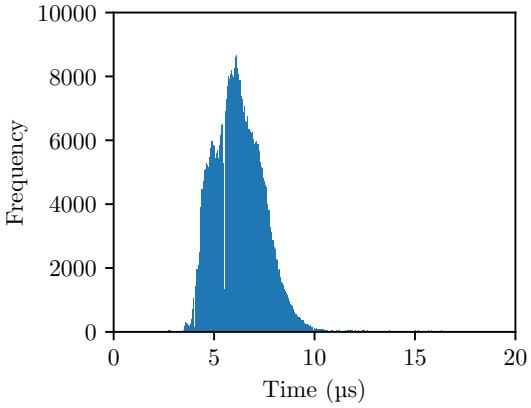


(a) Original ($n = 20$).

(b) Patched ($n = 20$).

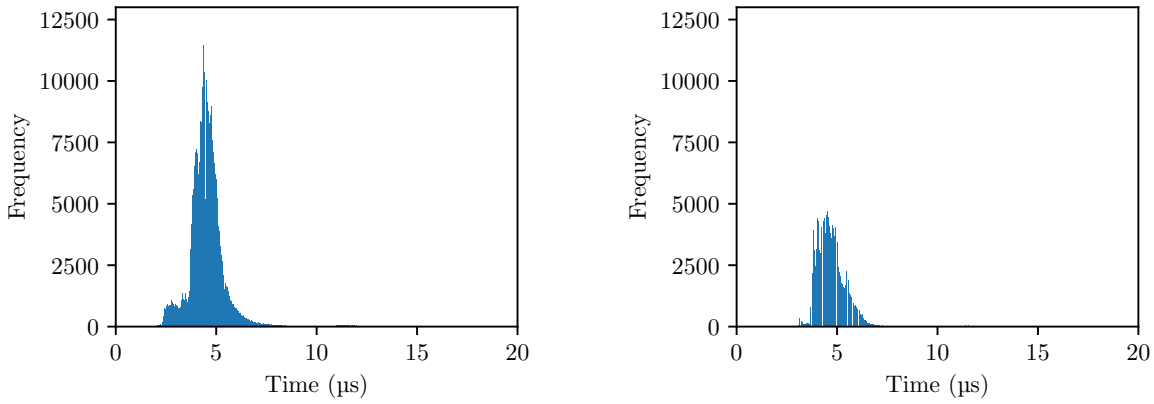


(c) Original ($n = 40$).



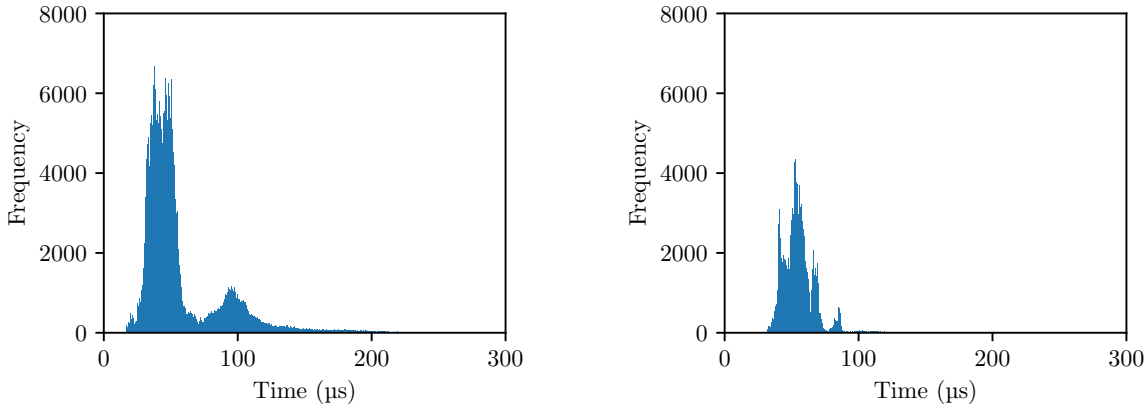
(d) Patched ($n = 40$).

Figure 5.23: `find_later_rq()` overhead.



(a) $n = 20$. (b) $n = 40$.

Figure 5.24: `check_wdl_preempt()` overhead.

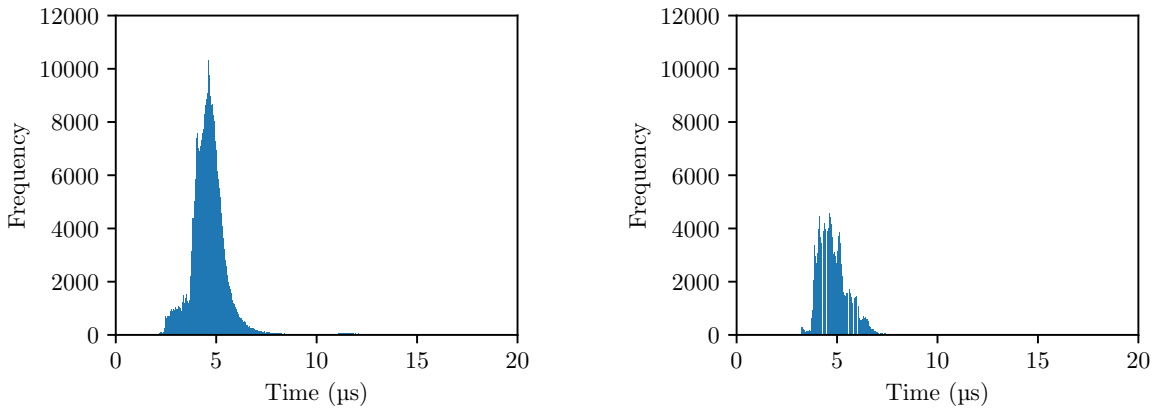


(a) $n = 20$. (b) $n = 40$.

Figure 5.25: `push_wdl_stop()` overhead.

This is not the case for `push_wdl_stop()`, which is called when `check_wdl_preempt()` discovers a violation of USE 1. Overheads for `push_wdl_stop()` are presented in Figure 5.25, in which a calls take on order 100 µs to complete. Not illustrated in Figure 5.25 are the long tails of these call duration distributions. The worst-case duration observed for `check_wdl_preempt()` was over a millisecond.

Note that these measurements start from when `check_wdl_preempt()` first discovers a violation and end when `push_wdl_stop()` returns after performing the relevant migration. As such, these measurements include the time needed to wake and schedule the `stop` task. This demonstrates that invoking the `stop` task to call `push_wdl_stop()` is expensive relative to operations such as enqueueing and dequeueing tasks. For workloads that do not require sub-millisecond granularity, these overheads may still be acceptable.



(a) $n = 20$. (b) $n = 40$.

Figure 5.26: `check_global_order()` overhead.

`check_global_order()` and `swap_global_stop()`. Overheads for function `check_global_order()` are presented in Figure 5.26 and overheads for function `swap_global_stop()` are presented in Figure 5.27. `check_global_order()` and `swap_global_stop()` are analogous to `check_wdl_preempt()` and `push_wdl_stop()` when it comes to overheads. `check_global_order()` is fast, while `swap_global_stop()` induces higher latency. The average duration of `swap_global_stop()` exceeds that of `push_wdl_stop()`, which is to be expected because `swap_global_stop()` must wake the `stop` tasks on two CPUs.

Note that Figure 5.27 does not contain a distribution for $n = 40$. As stated previously, when $n = 40$, our evaluated task systems each had at most one task in τ^{glob}. Because at least two tasks in τ^{glob} must be scheduled in order for USE 2 to be violated, `swap_global_stop()` was never called for such task systems.

Tardiness. Tardiness relative to tasks' periods is presented in Figure 5.28. The addition of our patch does not impact observed tardiness. This is expected because only a small minority of tasks are in τ^{glob}. Only these tasks are treated differently by our patched kernel.

Note that the tardiness presented in Figure 5.28 is much higher than that observed when evaluating our patch for IDENTICAL/SEMI-PARTITIONED (see Figure 5.7 in Section 5.2.6). This seems to be due to the ODROID-XU4's lack of computing capacity relative to the desktop machine used to evaluate the other patch. When the real-time workload consumes 95% of capacity, as is the case in this evaluation, the ODROID-XU4

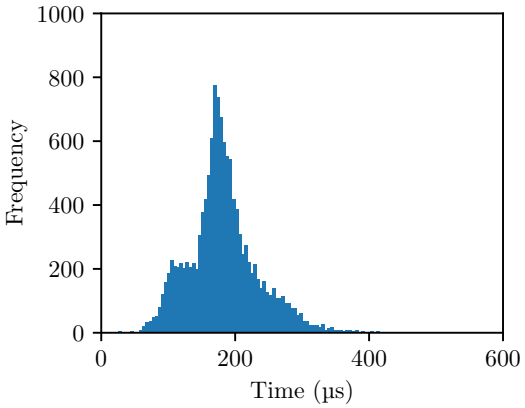
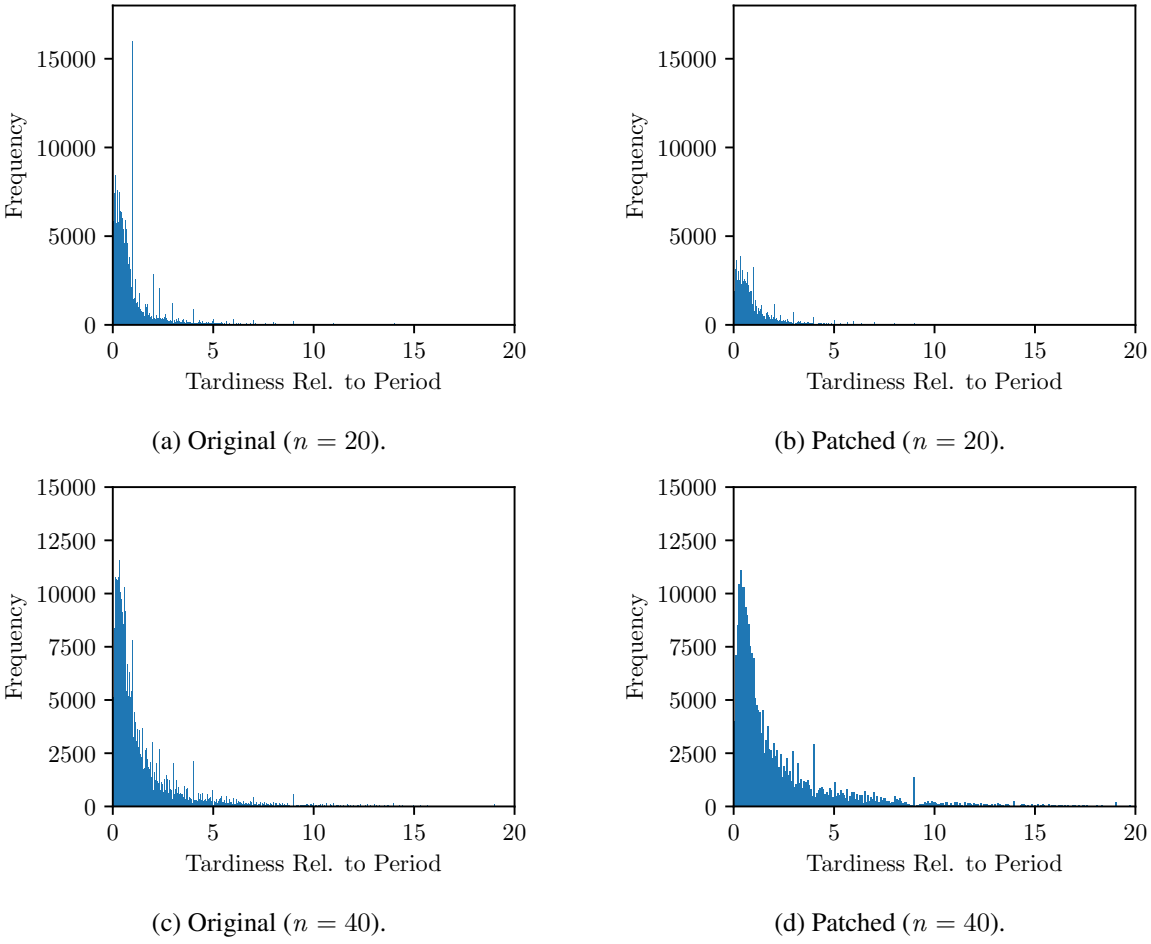


Figure 5.27: `swap_global_stop()` overhead ($n = 20$).



(a) Original ($n = 20$).

(b) Patched ($n = 20$).

(c) Original ($n = 40$).

(d) Patched ($n = 40$).

Figure 5.28: Relative tardiness.

begins to lag behind. This can delay the throttling and replenishment of SCHED_DEADLINE tasks, which causes increased tardiness.

5.4 Chapter Summary

In this chapter, we presented two patches to SCHED_DEADLINE aimed at restoring bounded response-time guarantees under special cases of heterogeneous multiprocessors. One patch targets IDENTICAL/SEMI-PARTITIONED systems, while the other targets a UNIFORM/SEMI-CLUSTERED system with two distinct speeds such that each task has affinity for either only the fast CPUs, only the slow CPUs, or all CPUs. We evaluated the increases in overheads due to our patches. While both patches increased overheads, such increases are acceptable for workloads not requiring sub-millisecond granularity.

CHAPTER 6: CONCLUSION

The behavior of EDF with respect to SRT remains poorly understood. This dissertation extends existing SRT-optimality results by considering heterogeneity and the broader class of WC schedulers. We have also experimented with implementing EDF variants targeting special cases of heterogeneous multiprocessors.

In this chapter, we summarize the results of this dissertation in Section 6.1, discuss other works to which the author contributed that were outside the scope of the dissertation in Section 6.2, acknowledge the contributions of coauthors in Section 6.3, and discuss future work in Section 6.4.

6.1 Summary of Results

Response-time bounds. In Chapter 3, we derived improved response-time bounds for Unr-WC on UNIFORM multiprocessors. We proved that Strong-APA-WC is SRT-optimal under IDENTICAL/ARBITRARY multiprocessors. We defined Unr-WC for UNRELATED multiprocessors, demonstrated that Unr-WC and Strong-APA-WC are special cases of Unr-WC, and proved response-time bounds under Unr-WC that asymptotically approach infinity as the task system and multiprocessor approach infeasibility.

Implementations. In Chapter 5, we presented two patches to `SCHED_DEADLINE`, Linux's EDF implementation. The goal of these patches is to guarantee bounded response times (assuming an idealized `SCHED_DEADLINE`) for special cases of heterogeneous multiprocessors. The first patch targets IDENTICAL/SEMI-PARTITIONED systems and the second patch targets UNIFORM/SEMI-CLUSTERED systems with two distinct speeds and affinities such that each task has affinity for either all fast CPUs, all slow CPUs, or all CPUs. Overheads increase as a result of these patches, but the increase seems acceptable so long as tasks do not require sub-millisecond granularity.

6.2 Other Publications

Other works done by the author concurrently with this dissertation but exist outside of its scope are concerned with providing *isolation* between tasks or between components comprised of tasks. Isolation mitigates

interference, which is the inflation of tasks' WCETs due to competition for shared resources such as caches, memory, accelerators, *etc.* Inflated WCETs may make it impossible to guarantee that real-time requirements are met. Isolation mitigates interference by partitioning shared resources across space (*e.g.*, concurrently running tasks are guaranteed exclusive access to distinct cache sets during their execution) and time (*e.g.*, two tasks that access the same cache sets are never scheduled concurrently).

Most of these works fall under the Mixed-Criticality on Multi-Core (MC^2) project (Anderson et al., 2009; Bakita et al., 2021; Chisholm et al., 2015, 2016, 2017; Kim et al., 2017, 2020; Ward et al., 2013). A goal of MC^2 is to provide varying degrees of isolation to tasks belonging to different *criticalities* (*e.g.*, failure of a high-criticality task may lead to loss of life, while failure of a low-criticality task may only lead to mild discomfort or inconvenience). Permitting a lower degree of isolation between low-criticality tasks allows systems to reap the throughput benefits of shared resources for low-criticality tasks without compromising the schedulability of high-criticality tasks. The author contributed to MC^2 by considering support for mode changes (Chisholm et al., 2017), OS features such as I/O and inter-process communication mechanisms (Kim et al., 2020), and simultaneous multithreading (Bakita et al., 2021). The author contributed to these works by the coding of schedulability studies, which evaluate the benefits of MC^2 by comparing the ratio of randomly generated synthetic task systems that can meet real-time guarantees under MC^2 against the ratio of systems when all tasks are isolated according to the highest criticality level.

Outside of the MC^2 project was a work by Voronov et al. (2021) concerning the sharing of accelerators between components (which are composed of tasks) while tasks may have interdependencies. These interdependencies are often represented as directed graphs such that nodes represent tasks and edges represent data dependencies. In such systems, the end-to-end response-time bound of a graph is of interest. This work considered merging nodes, *i.e.*, forcing jobs belonging to tasks of merged nodes to execute sequentially, for the purpose of reducing end-to-end response-time bounds. Merging reduces end-to-end bounds by removing analytical pessimism that scales with the length of the longest path in a graph; however, merging also reduces task-level parallelism, which can make the system unschedulable if merging is done too aggressively. This work presented heuristics for deciding what nodes to merge. The author contributed to coding the evaluation of these heuristics.

6.3 Acknowledgements

This dissertation would not have been possible without the collaboration of my co-authors. The analysis of $\mathcal{HP}$-$\mathcal{LAG}$ systems in Section 3.2 arose from discussion with Sergey Voronov. Sergey also assisted in the coding of the evaluation of Unr-WC in Section 3.5.3. Luca Abeni aided us in developing our understanding of the `SCHED_DEADLINE` implementation, which we used to design the patches discussed in Chapter 5.

6.4 Future Work

Unanswered questions about Unr-WC. As stated previously, simulations of periodic tasks under Unr-WC suggest that it may be SRT-optimal. Observed response times did not go to infinity as the slowdown factor of generated task systems went to 0, which is the behavior of the analytical bound in Theorem 3.36. Some modification to the analysis presented in Section 3.5.2, such as a more cleverly constructed invariant, may be necessary to demonstrate Unr-WC's SRT-optimality.

Tighter response-time bounds. As stated in Chapter 1, a well-known shortcoming of existing SRT analysis for EDF is that analytical response-time bounds are much larger than observed response times (outside of some constrained systems such as those with only a few processors Devi and Anderson (2008) or harmonic task systems Ahmed and Anderson (2021)). This remains true of the bounds proven in this dissertation for WC variants, of which EDF is a special case. A major limitation of our analysis is that deviation is constrained to be non-negative (recall Definition 3.2). Though this is necessary for the proofs, a consequence is that our analysis does not account for execution during which the virtual time exceeds the actual time. At a high level, this means our analysis permits jobs to not begin executing until they are already lagging behind. This does not match the behavior of EDF, under which (assuming a feasible system) a majority of jobs' execution generally occurs before their deadlines. Inability to account for such execution in our analysis is a major source of pessimism that may prevent the derivation of tighter bounds. The author conjectures that different abstractions are necessary to derive tighter bounds.

Implementing variants in an RTOS. Linux is attractive to develop on top of due to its popularity and richness of features. These points become drawbacks for formalization, as the frequent introduction of new features contribute to the kernel's size and complexity. The distributed runqueue structure in the scheduler is also difficult to reason about because of the lack of a consistent state agreed upon by all CPUs. This lack

of a consistent state can result in failed migrations, which is generally incompatible with real-time analysis. These drawback may not apply to an RTOS, as these are typically small and need not scale to as many CPUs as Linux. For example, in the Zephyr RTOS (Zephyr), all CPUs share a single runqueue (unless Zephyr is configured for PARTITIONED scheduling). Resources pertaining to symmetric multiprocessing are protected by a single lock. EDF scheduling is also supported. It may be possible to formally guarantee bounds under our considered EDF variants if implemented on an RTOS like Zephyr.

APPENDIX A: PROOF MODIFICATIONS FOR `SCHED_DEADLINE` PATCH

This appendix proves response-time bounds for the patched `SCHED_DEADLINE` detailed in Section 5.2 for IDENTICAL/SEMI-PARTITIONED. We reason about an idealized version of `SCHED_DEADLINE` where migrations are instantaneous. This ignores issues caused by synchronization such as failing to push a task from a runqueue due to a change in another runqueue's state.

The analysis in this appendix is fairly similar to that of $\mathcal{HP}$-$\mathcal{LAG}$ systems. Lemma A.3, to be proven in this appendix, is analogous to Lemma 3.14, which concerns $\mathcal{HP}$-$\mathcal{LAG}$ systems.

▽ **Definition A.1.** Consider $\tau' \subseteq \tau$. Let

$$\tau^{\mathrm{G}}(\tau') \triangleq \{\tau_i \in \tau' : \alpha_i = \pi\},$$
$$\tau^{\mathrm{P}}(\tau', \pi_j) \triangleq \{\tau_i \in \tau' : \alpha_i = \{\pi_j\}\},$$
$$\pi^{\mathrm{P}}(\tau') \triangleq \{\pi_j \in \pi : |\tau^{\mathrm{P}}(\tau', \pi_j)| > 0\},$$

i.e., $\tau^{\mathrm{G}}(\tau')$ denotes the subset of Global tasks in τ', $\tau^{\mathrm{P}}(\tau', \pi_j)$ denotes the subset of tasks in τ' that are Partitioned on processor π_j, and $\pi^{\mathrm{P}}(\tau')$ denotes the processors in π that have Partitioned tasks in τ'. △

▷ **Lemma A.1.** If a set of tasks τ' is such that $\mathcal{HP}(\tau', t^*)$ and $\tau' \subseteq \tau_{\text{rdy}}(t^*)$ for $t^* \in [t - C_{[1]}, t]$, then at time t, we have $\sum_{\tau_i \in \tau'} csp_i(t) \geq \min\{m, |\tau^{\mathrm{G}}(\tau')| + |\pi^{\mathrm{P}}(\tau')|\}$. ◁

Proof. Let an occupied processor be any processor whose runqueue contains a task in τ'.

► **Claim A.1.1.** The number of occupied processors is non-decreasing over $[t - C_{[1]}, t]$. ◄

Proof. The only way for an occupied processor to become unoccupied is for its last remaining task τ_i of τ' on its runqueue to be migrated to another runqueue. Because tasks of τ' are ready and have highest priority over $[t - C_{[1]}, t]$, task τ_i is scheduled on this (previously) occupied processor. While scheduled, task τ_i cannot be pulled. The only way for task τ_i to be migrated is to be pushed when returning from being throttled (recall our patch removes bypassing throttling). Under our patch, task τ_i can only be pushed to a processor whose runqueue only contains later-deadline tasks than the remaining tasks on the original runqueue. Because task τ_i is the last task of τ' on its original runqueue, any target runqueue being pushed to also has no tasks of τ'. Thus, the target runqueue

corresponds to an unoccupied processor that becomes occupied once task τ_i is pushed to it. Because, over $[t - C_{[1]}, t]$, whenever an occupied processor becomes unoccupied, an unoccupied processor must become occupied, the number of occupied processors is non-decreasing. ■

► **Claim A.1.2.** We can assume there is at least one unoccupied processor throughout $[t - C_{[1]}, t]$. ◄

Proof. Suppose the assumption is false such that there are time instants in $[t - C_{[1]}, t]$ such that there are no unoccupied processors. By Claim A.1.1, there are no unoccupied processors at time t. Then each processor has a task of τ' on its runqueue. Because tasks of τ' are ready and have highest priority at time t, each processor schedules a task of τ' at time t. Thus, we would have $\sum_{\tau_i \in \tau'} csp_i(t) = m$, which satisfies the lemma statement. Because the assumption being false yields the lemma, we can assume for the remainder of the proof that the assumption is true. ■

► **Claim A.1.3.** All tasks of $\tau^{\mathrm{G}}(\tau')$ are scheduled throughout $[t - C_{[1]}, t]$. ◄

Proof. Suppose otherwise that there is simultaneously an unoccupied processor (by the assumption in Claim A.1.2) and an unscheduled task in $\tau^{\mathrm{G}}(\tau')$ with affinity for said processor (tasks in $\tau^{\mathrm{G}}(\tau')$ have affinity for all processors in the `root_domain`). This contradicts that `SCHED_DEADLINE` would have scheduled such a task. ■

► **Claim A.1.4.** At time t, every processor $\pi_j \in \pi^{\mathrm{P}}(\tau')$ schedules a task $\tau_i \in \tau^{\mathrm{P}}(\tau', \pi_j)$. ◄

Proof. Over $[t - C_{[1]}, t]$, a processor $\pi_j \in \pi^{\mathrm{P}}(\tau')$ will never pull a task in $\tau^{\mathrm{G}}(\tau')$ because, by Claim A.1.3, tasks in $\tau^{\mathrm{G}}(\tau')$ are already scheduled. Also over $[t - C_{[1]}, t]$, a task in $\tau^{\mathrm{G}}(\tau')$ will never be pushed to processor π_j because, by Claim A.1.2, there is always an unoccupied processor that will be pushed to instead of processor π_j. Thus, the only way $\pi_j \in \pi^{\mathrm{P}}(\tau')$ may not schedule a task $\tau_i \in \tau^{\mathrm{P}}(\tau', \pi_j)$ at time t is if there is a higher-priority task $\tau_k \in \tau^{\mathrm{G}}(\tau')$ scheduled on π_j over $[t - C_{[1]}, t]$. By Claim A.1.3, there is at most one $\tau_k \in \tau^{\mathrm{G}}(\tau')$ on π_j's runqueue over $[t - C_{[1]}, t]$. Because $C_k \leq C_{[1]}$, task τ_k must finish at least one job in $[t - C_{[1]}, t]$. Thus, τ_k is throttled and replenished in $[t - C_{[1]}, t]$. When τ_k is replenished, it will be pushed to an unoccupied processor, after which some task $\tau_i \in \tau^{\mathrm{P}}(\tau', \pi_j)$ will be scheduled until at least time t. ■

By Claims A.1.3 and A.1.4, there are at least $\left|\tau^{\mathrm{G}}(\tau')\right| + \left|\pi^{\mathrm{P}}(\tau')\right|$ scheduled tasks of τ' at time t. Thus, $\sum_{\tau_i \in \tau'} csp_i(t) \geq \left|\tau^{\mathrm{G}}(\tau')\right| + \left|\pi^{\mathrm{P}}(\tau')\right|$. This satisfies the lemma statement. □

▷ **Lemma A.2.** For any time t and task set $\tau' \subseteq \tau$,

$$\forall t^* \leq t : \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i(t^*) \geq \left(\sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i(t) \right) - U\left(\tau'\right) \cdot (t - t^*) . \qquad \triangleleft$$

Proof. Consider a single task $\tau_i \in \tau'$. We have

$$\left.\begin{aligned}
\sqrt{u_i} \cdot dev_i(t) - u_i \cdot (t - t^*) = {} & \{\text{Definition 3.2}\} \\
& \sqrt{u_i} \cdot \max\left\{0, \sqrt{u_i} \cdot (t - vt_i(t))\right\} - u_i \cdot (t - t^*) \\
= {} & \max\left\{0, u_i \cdot (t - vt_i(t))\right\} - u_i \cdot (t - t^*) \\
= {} & \max\left\{-u_i \cdot (t - t^*), u_i \cdot (t^* - vt_i(t))\right\} \\
\leq {} & \{t - t^* \geq 0\} \\
& \max\left\{0, u_i \cdot (t^* - vt_i(t))\right\} \\
\leq {} & \{\text{Lemma 3.6}\} \\
& \max\left\{0, u_i \cdot (t^* - vt_i(t^*))\right\} \\
= {} & \sqrt{u_i} \cdot \max\left\{0, \sqrt{u_i} \cdot (t^* - vt_i(t^*))\right\} \\
= {} & \{\text{Definition 3.2}\} \\
& \sqrt{u_i} \cdot dev_i(t^*).
\end{aligned}\right\} \qquad (A.1)$$

Summing (A.1) over the tasks in τ' yields the lemma. □

▽ **Definition A.2.** For each subset $\tau' \subseteq \tau$, let

$$\beta^{\mathrm{DL}}_{\tau'} \triangleq \left(T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}} \right) \frac{U(\tau')}{2u_{[n]}} \left(2U_{\max} - U\left(\tau'\right)\right) . \qquad \triangle$$

▷ **Lemma A.3.** Let $[t_0, t_1)$ be a time interval such that

$$\exists \tau^{\text{const}} \subseteq \tau : \ \forall t \in [t_0, t_1) : \ \tau_{\text{act}}(t) = \tau^{\text{const}}$$

and at time t_0, we have

$$\forall t \leq t_0 : \forall \tau' \subseteq \tau_{\text{act}}(t) : \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i(t) \leq \beta^{\text{DL}}_{\tau'}. \tag{A.2}$$

We have

$$\forall \tau' \subseteq \tau^{\text{const}} : \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i(t) \leq \beta^{\text{DL}}_{\tau'}. \tag{A.3}$$

for any $t \in [t_0, t_1)$. ◁

Proof. We prove by contradiction. Suppose otherwise that there exist time instants in $[t_0, t_1)$ such that (A.3) is false. By (A.2), (A.3) is true at time t_0. Let $t_{\text{b}} \in [t_0, t_1)$ denote the latest time instant such that (A.3) is true over $[t_0, t_{\text{b}})$. We will show that the existence of t_{b} leads to a contradiction.

► **Claim A.3.1.** $\forall t \leq t_{\text{b}} : \forall \tau' \subseteq \tau_{\text{act}}(t) : \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i(t) \leq \beta^{\text{DL}}_{\tau'}$. ◄

Proof. By the definition of t_{b}, we have that

$$\forall t \in [t_0, t_b) : \forall \tau' \subseteq \tau^{\text{const}} : \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i(t) \leq \beta^{\text{DL}}_{\tau'}. \tag{A.4}$$

Thus, at time t_{b}, for any $\tau' \subseteq \tau^{\text{const}}$, we have

$$\left.\begin{aligned} \beta^{\text{DL}}_{\tau'} &\geq \{\text{Equation (A.4)}\} \\ &\quad \lim_{t^* \to t_{\text{b}}^-} \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i(t^*) \\ &= \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot \lim_{t^* \to t_{\text{b}}^-} dev_i(t^*) \\ &\geq \{\text{Lemma 3.10}\} \\ &\quad \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i(t_{\text{b}}). \end{aligned}\right\} \tag{A.5}$$

The claim follows from (A.2), (A.4), and (A.5). ■

► **Claim A.3.2.** At time t_{b}, there exists $\tau^{\mathrm{b}} \subseteq \tau^{\mathrm{const}}$ such that both

$$\forall \psi > 0 : \exists t^* \in (t_{\mathrm{b}}, t_{\mathrm{b}} + \psi) : \sum_{\tau_i \in \tau^{\mathrm{b}}} \sqrt{u_i} \cdot dev_i(t^*) > \beta^{\mathrm{DL}}_{\tau^{\mathrm{b}}} \text{ and} \tag{A.6}$$

$$\sum_{\tau_i \in \tau^{\mathrm{b}}} \sqrt{u_i} \cdot dev_i(t_{\mathrm{b}}) = \beta^{\mathrm{DL}}_{\tau^{\mathrm{b}}} \tag{A.7}$$

are true. ◄

Proof. First prove (A.6) by contradiction. Suppose otherwise that

$$\forall \tau^{\mathrm{b}} \subseteq \tau^{\mathrm{const}} : \exists \psi > 0 : \forall t \in (t_{\mathrm{b}}, t_{\mathrm{b}} + \psi) : \sum_{\tau_i \in \tau^{\mathrm{b}}} \sqrt{u_i} \cdot dev_i(t) \leq \beta^{\mathrm{DL}}_{\tau^{\mathrm{b}}}.$$

Because t_{b} is defined such that (A.3) is true over $[t_0, t_{\mathrm{b}})$ and $[t_0, t_{\mathrm{b}}) \cup (t_{\mathrm{b}}, t_{\mathrm{b}} + \psi) = [t_0, t_{\mathrm{b}} + \psi)$, we have

$$\forall \tau^{\mathrm{b}} \subseteq \tau^{\mathrm{const}} : \forall \in [t_0, t_{\mathrm{b}} + \psi) : \sum_{\tau_i \in \tau^{\mathrm{b}}} \sqrt{u_i} \cdot dev_i(t^*) \leq \beta^{\mathrm{DL}}_{\tau^{\mathrm{b}}}.$$

This contradicts the definition of t_{b} as the latest time instant such that (A.3) is true over $[t_0, t_{\mathrm{b}})$.

It remains to prove (A.7). We have

$$\begin{aligned}
\sum_{\tau_i \in \tau^{\mathrm{b}}} \sqrt{u_i} \cdot dev_i(t_{\mathrm{b}}) &= \{\text{Corollary 3.12}\} \\
&\quad \sum_{\tau_i \in \tau^{\mathrm{b}}} \sqrt{u_i} \cdot \lim_{t^* \to t_{\mathrm{b}}^+} dev_i(t^*) \\
&= \lim_{t^* \to t_{\mathrm{b}}^+} \sum_{\tau_i \in \tau^{\mathrm{b}}} \sqrt{u_i} \cdot dev_i(t^*) \\
&\geq \{\text{Equation (A.6)}\} \\
&\quad \beta^{\mathrm{DL}}_{\tau^{\mathrm{b}}}.
\end{aligned}$$

By Claim A.3.1, we have

$$\sum_{\tau_i \in \tau^{\mathrm{b}}} \sqrt{u_i} \cdot dev_i(t_{\mathrm{b}}) \leq \beta^{\mathrm{DL}}_{\tau^{\mathrm{b}}}.$$

Because $\sum_{\tau_i \in \tau^{\mathrm{b}}} \sqrt{u_i} \cdot dev_i(t_{\mathrm{b}})$ is both upper and lower bounded by $\beta^{\mathrm{DL}}_{\tau^{\mathrm{b}}}$, (A.7) is the only possibility. This completes the proof of the claim. ■

► **Claim A.3.3.** $\forall t \in [t_{\mathrm{b}} - C_{[1]}, t_{\mathrm{b}}] : \forall \tau_e \in \tau^{\mathrm{b}} :$

$$\frac{dev_e(t)}{\sqrt{u_e}} \geq \frac{T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}}}{2u_{[n]}} \left(2U_{\max} - 2U\left(\tau^{\mathrm{b}}\right) + u_e\right) - \frac{U\left(\tau^{\mathrm{b}}\right)}{u_e} \cdot (t_{\mathrm{b}} - t)\,. \qquad \blacktriangleleft$$

Proof. We have

$$\begin{aligned}
&\sqrt{u_e} \cdot dev_e(t) \\
&= \sum_{\tau_i \in \tau^{\mathrm{b}}} \sqrt{u_i} \cdot dev_i(t) - \sum_{\tau_i \in \tau^{\mathrm{b}} \setminus \{\tau_e\}} \sqrt{u_i} \cdot dev_i(t) \\
&\geq \{\text{Lemma A.2}\} \\
&\quad \sum_{\tau_i \in \tau^{\mathrm{b}}} \sqrt{u_i} \cdot dev_i(t_{\mathrm{b}}) - U\left(\tau^{\mathrm{b}}\right) \cdot (t_{\mathrm{b}} - t) - \sum_{\tau_i \in \tau^{\mathrm{b}} \setminus \{\tau_e\}} \sqrt{u_i} \cdot dev_i(t) \\
&= \{\text{Claim A.3.2, Equation (A.7)}\} \\
&\quad \beta^{\mathrm{DL}}_{\tau^{\mathrm{b}}} - U\left(\tau^{\mathrm{b}}\right) \cdot (t_{\mathrm{b}} - t) - \sum_{\tau_i \in \tau^{\mathrm{b}} \setminus \{\tau_e\}} \sqrt{u_i} \cdot dev_i(t) \\
&= \{\text{Lemma 3.5}\} \\
&\quad \beta^{\mathrm{DL}}_{\tau^{\mathrm{b}}} - U\left(\tau^{\mathrm{b}}\right) \cdot (t_{\mathrm{b}} - t) - \sum_{\tau_i \in (\tau^{\mathrm{b}} \setminus \{\tau_e\}) \cap \tau_{\text{act}}(t)} \sqrt{u_i} \cdot dev_i(t) \\
&\geq \{\text{Claim A.3.1}\} \\
&\quad \beta^{\mathrm{DL}}_{\tau^{\mathrm{b}}} - U\left(\tau^{\mathrm{b}}\right) \cdot (t_{\mathrm{b}} - t) - \beta^{\mathrm{DL}}_{(\tau^{\mathrm{b}} \setminus \{\tau_e\}) \cap \tau_{\text{act}}(t)} \\
&\geq \left\{\text{Definition A.2, Lemma 3.15, and } \left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right) \subset \tau^{\mathrm{b}} \subseteq \tau_{\text{act}}(t_{\mathrm{b}})\right\} \\
&\quad \beta^{\mathrm{DL}}_{\tau^{\mathrm{b}}} - U\left(\tau^{\mathrm{b}}\right) \cdot (t_{\mathrm{b}} - t) - \beta^{\mathrm{DL}}_{\tau^{\mathrm{b}} \setminus \{\tau_e\}} \\
&= \{\text{Definition A.2}\}
\end{aligned}$$

$$
\begin{aligned}
&\frac{T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}}}{2u_{[n]}} \left(U\left(\tau^{\mathrm{b}}\right)\right) \left(2U_{\max} - U\left(\tau^{\mathrm{b}}\right)\right) - U\left(\tau^{\mathrm{b}}\right) \cdot (t_{\mathrm{b}} - t) \\
&\quad - \frac{T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}}}{2u_{[n]}} \left(U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right)\right) \left(2U_{\max} - U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right)\right) \\
= \; &\left\{\text{By Definition 2.15, } U\left(\tau^{\mathrm{b}}\right) = U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right) + u_e\right\} \\
&\frac{T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}}}{2u_{[n]}} \left(U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right) + u_e\right) \left(2U_{\max} - U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right) - u_e\right) - U\left(\tau^{\mathrm{b}}\right) \cdot (t_{\mathrm{b}} - t) \\
&\quad - \frac{T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}}}{2u_{[n]}} \left(U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right)\right) \left(2U_{\max} - U\left(\tau^{\mathrm{b}} \setminus \{\tau_e\}\right)\right) \\
= \; &\frac{T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}}}{2u_{[n]}} \left(2U_{\max} - 2U\left(\tau^{\mathrm{b}}\right) + u_e\right) u_e - U\left(\tau^{\mathrm{b}}\right) \cdot (t_{\mathrm{b}} - t) \,.
\end{aligned}
$$

Dividing by u_e yields the claim. ■

► **Claim A.3.4.** $\forall \tau_e \in \tau^{\mathrm{b}} : \; \forall t \in [t_{\mathrm{b}} - C_{[1]}, t_{\mathrm{b}}] : \; dev_e(t) > 0$ and $\tau_e \in \tau_{\mathrm{rdy}}(t)$. ◄

Proof. For any $\tau_e \in \tau^{\mathrm{b}}$, we have

$$
\begin{aligned}
\sqrt{u_e} \cdot dev_e(t) \geq \; &\{\text{Claim A.3.3}\} \\
&\frac{T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}}}{2u_{[n]}} \left(2U_{\max} - 2U\left(\tau^{\mathrm{b}}\right) + u_e\right) u_e - U\left(\tau^{\mathrm{b}}\right) \cdot (t_{\mathrm{b}} - t) \\
\geq \; &\left\{U\left(\tau^{\mathrm{b}}\right) \leq U_{\max}\right\} \\
&\frac{T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}}}{2u_{[n]}} u_e^2 - U\left(\tau^{\mathrm{b}}\right) \cdot (t_{\mathrm{b}} - t) \\
> \; &\left\{T_{[1]} > 0\right\} \\
&\frac{m \cdot C_{[1]}}{u_{[n]}^2} u_e^2 - U\left(\tau^{\mathrm{b}}\right) \cdot (t_{\mathrm{b}} - t) \\
\geq \; &\left\{u_e \geq u_{[n]}\right\} \\
&m \cdot C_{[1]} - U\left(\tau^{\mathrm{b}}\right) \cdot (t_{\mathrm{b}} - t) \\
\geq \; &\left\{t \geq t_{\mathrm{b}} - C_{[1]}\right\} \\
&\left(m - U\left(\tau^{\mathrm{b}}\right)\right) (t_{\mathrm{b}} - t) \\
\geq \; &\left\{t_{\mathrm{b}} \geq t \text{ and } m \geq U\left(\tau^{\mathrm{b}}\right)\right\}
\end{aligned}
$$

0.

The claim follows from Lemma 3.1. ■

► **Claim A.3.5.** $\forall t \in [t_{\mathrm{b}} - C_{[1]}, t_{\mathrm{b}}] : \forall \tau_\ell \in \tau_{\mathrm{act}}(t) \setminus \tau^{\mathrm{b}} :$

$$\frac{dev_\ell(t)}{\sqrt{u_e}} \leq \frac{T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}}}{2u_{[n]}} \left(2U_{\max} - 2U\left(\tau^{\mathrm{b}}\right) - u_\ell\right) + \frac{U\left(\tau^{\mathrm{b}}\right)}{u_\ell} \left(t_{\mathrm{b}} - t\right). \quad ◄$$

Proof. We have

$$\begin{aligned}
&\sqrt{u_\ell} \cdot dev_\ell(t) \\
&= \sum_{\tau_i \in \tau^{\mathrm{b}} \cup \{\tau_\ell\}} \sqrt{u_i} \cdot dev_i(t) - \sum_{\tau_i \in \tau^{\mathrm{b}}} \sqrt{u_i} \cdot dev_i(t) \\
&\leq \{\text{Lemma A.2}\} \\
&\quad \sum_{\tau_i \in \tau^{\mathrm{b}} \cup \{\tau_\ell\}} \sqrt{u_i} \cdot dev_i(t) - \sum_{\tau_i \in \tau^{\mathrm{b}}} \sqrt{u_i} \cdot dev_i(t_{\mathrm{b}}) + U\left(\tau^{\mathrm{b}}\right) \cdot (t_{\mathrm{b}} - t) \\
&= \{\text{Claim A.3.2, Equation (A.7)}\} \\
&\quad \sum_{\tau_i \in \tau^{\mathrm{b}} \cup \{\tau_\ell\}} \sqrt{u_i} \cdot dev_i(t) - \beta^{\mathrm{DL}}_{\tau^{\mathrm{b}}} + U\left(\tau^{\mathrm{b}}\right) \cdot (t_{\mathrm{b}} - t) \\
&= \{\text{Lemma 3.5}\} \\
&\quad \sum_{\tau_i \in (\tau^{\mathrm{b}} \cup \{\tau_\ell\}) \cap \tau_{\mathrm{act}}(t)} \sqrt{u_i} \cdot dev_i(t) - \beta^{\mathrm{DL}}_{\tau^{\mathrm{b}}} + U\left(\tau^{\mathrm{b}}\right) \cdot (t_{\mathrm{b}} - t) \\
&\leq \{\text{Claim A.3.1}\} \\
&\quad \beta^{\mathrm{DL}}_{(\tau^{\mathrm{b}} \cup \{\tau_\ell\}) \cap \tau_{\mathrm{act}}(t)} - \beta^{\mathrm{DL}}_{\tau^{\mathrm{b}}} + U\left(\tau^{\mathrm{b}}\right) \cdot (t_{\mathrm{b}} - t) \\
&= \left\{\tau_\ell \in \tau_{\mathrm{act}}(t) \text{ and, by Claim A.3.4, } \tau^{\mathrm{b}} \subseteq \tau_{\mathrm{act}}(t)\right\} \\
&\quad \beta^{\mathrm{DL}}_{\tau^{\mathrm{b}} \cup \{\tau_\ell\}} - \beta^{\mathrm{DL}}_{\tau^{\mathrm{b}}} + U\left(\tau^{\mathrm{b}}\right) \cdot (t_{\mathrm{b}} - t) \\
&= \{\text{Definition A.2}\} \\
&\quad \frac{T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}}}{2u_{[n]}} \left(U\left(\tau^{\mathrm{b}} \cup \{\tau_\ell\}\right)\right) \left(2U_{\max} - U\left(\tau^{\mathrm{b}} \cup \{\tau_\ell\}\right)\right) \\
&\quad - \frac{T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}}}{2u_{[n]}} \left(U\left(\tau^{\mathrm{b}}\right)\right) \left(2U_{\max} - U\left(\tau^{\mathrm{b}}\right)\right) + U\left(\tau^{\mathrm{b}}\right) \cdot (t_{\mathrm{b}} - t)
\end{aligned}$$

$$
\begin{aligned}
&= \left\{\text{By Definition 2.15, } U\left(\tau^{\mathsf{b}} \cup \{\tau_\ell\}\right) = U\left(\tau^{\mathsf{b}}\right) + u_\ell\right\} \\
&\quad \frac{T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}}}{2u_{[n]}} \left(U\left(\tau^{\mathsf{b}}\right) + u_\ell\right) \left(2U_{\max} - U\left(\tau^{\mathsf{b}}\right) - u_\ell\right) \\
&\quad - \frac{T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}}}{2u_{[n]}} \left(U\left(\tau^{\mathsf{b}}\right)\right) \left(2U_{\max} - U\left(\tau^{\mathsf{b}}\right)\right) + U\left(\tau^{\mathsf{b}}\right) \cdot (t_{\mathsf{b}} - t) \\
&= \frac{T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}}}{2u_{[n]}} \left(2U_{\max} - 2U\left(\tau^{\mathsf{b}}\right) - u_\ell\right) u_\ell + U\left(\tau^{\mathsf{b}}\right) \cdot (t_{\mathsf{b}} - t).
\end{aligned}
$$

Dividing by u_ℓ yields the claim. ■

► **Claim A.3.6.** $\forall t \in [t_{\mathsf{b}} - C_{[1]}, t_{\mathsf{b}}] :\ \mathcal{HP}\left(\tau^{\mathsf{b}}, t\right)$. ◄

Proof. For any $t \in [t_{\mathsf{b}} - C_{[1]}, t_{\mathsf{b}}]$, consider any task $\tau_e \in \tau^{\mathsf{b}}$ and $\tau_\ell \in \tau_{\text{act}}(t) \setminus \tau^{\mathsf{b}}$. We have

$$
\begin{aligned}
&\frac{dev_e(t)}{\sqrt{u_e}} \\
&\geq \{\text{Claim A.3.3}\} \\
&\quad \frac{T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}}}{2u_{[n]}} \left(2U_{\max} - 2U\left(\tau^{\mathsf{b}}\right) + u_e\right) - \frac{U\left(\tau^{\mathsf{b}}\right)}{u_e} (t_{\mathsf{b}} - t) \\
&= \frac{T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}}}{2u_{[n]}} \left(2U_{\max} - 2U\left(\tau^{\mathsf{b}}\right)\right) + \frac{T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}}}{2u_{[n]}} u_e - \frac{U\left(\tau^{\mathsf{b}}\right)}{u_e} (t_{\mathsf{b}} - t) \\
&= \frac{T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}}}{2u_{[n]}} \left(2U_{\max} - 2U\left(\tau^{\mathsf{b}}\right) - u_\ell\right) + \frac{U\left(\tau^{\mathsf{b}}\right)}{u_\ell} (t_{\mathsf{b}} - t) \\
&\quad + \frac{T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}}}{2u_{[n]}} (u_e + u_\ell) - U\left(\tau^{\mathsf{b}}\right) \left(\frac{1}{u_e} + \frac{1}{u_\ell}\right) (t_{\mathsf{b}} - t) \\
&\geq \{\text{Claim A.3.5}\} \\
&\quad \frac{dev_\ell(t)}{\sqrt{u_\ell}} + \frac{T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}}}{2u_{[n]}} (u_e + u_\ell) - U\left(\tau^{\mathsf{b}}\right) \left(\frac{1}{u_e} + \frac{1}{u_\ell}\right) (t_{\mathsf{b}} - t) \\
&\geq \left\{u_e + u_\ell \geq 2u_{[n]}\right\} \\
&\quad \frac{dev_\ell(t)}{\sqrt{u_\ell}} + T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}} - U\left(\tau^{\mathsf{b}}\right) \left(\frac{1}{u_e} + \frac{1}{u_\ell}\right) (t_{\mathsf{b}} - t) \\
&\geq \left\{\frac{2}{u_{[n]}} \geq \frac{1}{u_e} + \frac{1}{u_\ell},\ m \geq U\left(\tau^{\mathsf{b}}\right), \text{ and } C_{[1]} \geq t_{\mathsf{b}} - t\right\} \\
&\quad \frac{dev_\ell(t)}{\sqrt{u_\ell}} + T_{[1]}.
\end{aligned}
$$

By Lemma 3.3 (recall that with $pp_i(t) = d_i(t)$, by Definition 2.20, $\phi = 0$), we have $pp_e(t_\mathrm{b}) < pp_\ell(t_\mathrm{b})$. The claim follows by Definition 3.3. ■

► **Claim A.3.7.** $\sum_{\tau_i \in \tau^\mathrm{b}} csp_i(t_\mathrm{b}) \geq U(\tau^\mathrm{b})$. ◄

Proof. By Claim A.3.4, any task in τ^b is ready at any $t \in [t_\mathrm{b} - C_{[1]}, t_\mathrm{b}]$, *i.e.*, $\tau^\mathrm{b} \subseteq \tau_\mathrm{rdy}(t)$. Additionally, by Claim A.3.6, tasks in τ^b have the highest priorities at any $t \in [t_\mathrm{b} - C_{[1]}, t_\mathrm{b}]$, *i.e.*, $\mathcal{HP}(\tau^\mathrm{b}, t)$. By Lemma A.1, we have $\sum_{\tau_i \in \tau^\mathrm{b}} csp_i(t_\mathrm{b}) \geq \min\{m, |\tau^\mathrm{G}(\tau^\mathrm{b})| + |\pi^\mathrm{P}(\tau^\mathrm{b})|\}$. At least one of the following cases must apply of $\sum_{\tau_i \in \tau^\mathrm{b}} csp_i(t_\mathrm{b})$.

◄ **Case A.3.1.** $\sum_{\tau_i \in \tau^\mathrm{b}} csp_i(t_\mathrm{b}) \geq m$. ►

We have

$$
\begin{aligned}
\sum_{\tau_i \in \tau^\mathrm{b}} csp_i(t_\mathrm{b}) &\geq m \\
&= \sum_{\pi_j \in \pi} 1.0 \\
&= \Big\{\text{Under IDENTICAL, capacity } sp^{(j)} \text{ is } 1.0\Big\} \\
&\quad \sum_{\pi_j \in \pi} sp^{(j)} \\
&\geq \left\{\text{Under ACS, } \frac{\texttt{sched\_rt\_runtime\_us}}{\texttt{sched\_rt\_period\_us}} \leq 1.0\right\} \\
&\quad \frac{\texttt{sched\_rt\_runtime\_us}}{\texttt{sched\_rt\_period\_us}} \cdot \sum_{\pi_j \in \pi} sp^{(j)} \\
&\geq \{\text{Equation (4.1) of ACS}\} \\
&\quad \sum_{\tau_i \in \tau_\mathrm{act}(t_\mathrm{b})} u_i \\
&\geq \{\tau^\mathrm{b} \subseteq \tau_\mathrm{act}(t_\mathrm{b})\} \\
&\quad \sum_{\tau_i \in \tau^\mathrm{b}} u_i \\
&= \{\text{Definition 2.15}\} \\
&\quad U(\tau^\mathrm{b}).
\end{aligned}
$$

♦

◀ **Case A.3.2.** $\sum_{\tau_i \in \tau^{\mathrm{b}}} csp_i(t_{\mathrm{b}}) \geq \left|\tau^{\mathrm{G}}\left(\tau^{\mathrm{b}}\right)\right| + \left|\pi^{\mathrm{P}}\left(\tau^{\mathrm{b}}\right)\right|$. ▶

We have

$$
\begin{aligned}
& \sum_{\tau_i \in \tau^{\mathrm{b}}} csp_i(t_{\mathrm{b}}) \\
\geq\ & \left|\tau^{\mathrm{G}}\left(\tau^{\mathrm{b}}\right)\right| + \left|\pi^{\mathrm{P}}\left(\tau^{\mathrm{b}}\right)\right| \\
=\ & \sum_{\tau_i \in \tau^{\mathrm{G}}(\tau^{\mathrm{b}})} 1 + \left|\pi^{\mathrm{P}}\left(\tau^{\mathrm{b}}\right)\right| \\
\geq\ & \{\text{Under } \texttt{SCHED\_DEADLINE}\text{, each } u_i \leq 1.0\} \\
& \sum_{\tau_i \in \tau^{\mathrm{G}}(\tau^{\mathrm{b}})} u_i + \left|\pi^{\mathrm{P}}\left(\tau^{\mathrm{b}}\right)\right| \\
=\ & \sum_{\tau_i \in \tau^{\mathrm{G}}(\tau^{\mathrm{b}})} u_i + \sum_{\pi_j \in \pi^{\mathrm{P}}(\tau^{\mathrm{b}})} 1 \\
\geq\ & \left\{\text{Under ACS, } \frac{\texttt{sched\_rt\_runtime\_us}}{\texttt{sched\_rt\_period\_us}} \leq 1.0\right\} \\
& \sum_{\tau_i \in \tau^{\mathrm{G}}(\tau^{\mathrm{b}})} u_i + \sum_{\pi_j \in \pi^{\mathrm{P}}(\tau^{\mathrm{b}})} \frac{\texttt{sched\_rt\_runtime\_us}}{\texttt{sched\_rt\_period\_us}} \\
\geq\ & \{\text{Equation (5.1) of patched ACS}\} \\
& \sum_{\tau_i \in \tau^{\mathrm{G}}(\tau^{\mathrm{b}})} u_i + \sum_{\pi_j \in \pi^{\mathrm{P}}(\tau^{\mathrm{b}})} \left(\sum_{\tau_i:\ \alpha_i = \{\pi_j\}} u_i \right) \\
=\ & \left\{\text{Under SEMI-PARTITIONED, } \tau_i \in \tau^{\mathrm{G}}\left(\tau^{\mathrm{b}}\right) \text{ or } \alpha_i = \{\pi_j\} \text{ for some } \pi_j \in \pi^{\mathrm{P}}\left(\tau^{\mathrm{b}}\right)\right\} \\
& \sum_{\tau_i \in \tau^{\mathrm{b}}} u_i \\
=\ & \{\text{Definition 2.15}\} \\
& U\left(\tau^{\mathrm{b}}\right).
\end{aligned}
$$

♦

In either case, we have $\sum_{\tau_i \in \tau^{\mathrm{b}}} csp_i(t_{\mathrm{b}}) \geq U\left(\tau^{\mathrm{b}}\right)$. ■

By Lemma 3.13 and Claim A.3.4, for any task $\tau_e \in \tau^{\mathrm{b}}$, there exists $\psi > 0$ such that $\forall t \in [t_{\mathrm{b}}, t_{\mathrm{b}} + \psi)$:

$$\sqrt{u_e} \cdot dev_e(t) \leq \sqrt{u_e} \cdot dev_e(t_{\mathrm{b}}) + (t - t_{\mathrm{b}}) \cdot (u_i - csp_e(t_{\mathrm{b}}))\,.$$

Summing over the tasks in τ^{b}, we have $\forall t \in [t_{\mathrm{b}}, t_{\mathrm{b}} + \psi)$:

$$\begin{aligned}
\sum_{\tau_e \in \tau^{\mathrm{b}}} \sqrt{u_e} \cdot dev_e(t) &\leq \sum_{\tau_e \in \tau^{\mathrm{b}}} \sqrt{u_e} \cdot dev_e(t_{\mathrm{b}}) + (t - t_{\mathrm{b}}) \cdot (u_i - csp_e(t_{\mathrm{b}})) \\
&= \left[\sum_{\tau_e \in \tau^{\mathrm{b}}} \sqrt{u_e} \cdot dev_e(t_{\mathrm{b}}) \right] + (t - t_{\mathrm{b}}) \left[\sum_{\tau_e \in \tau^{\mathrm{b}}} (u_i - csp_e(t_{\mathrm{b}})) \right] \\
&= \{\text{Claim A.3.2, Equation (A.7)}\} \\
&\quad \beta^{\mathtt{DL}}_{\tau^{\mathrm{b}}} + (t - t_{\mathrm{b}}) \left[\sum_{\tau_e \in \tau^{\mathrm{b}}} (u_i - csp_e(t_{\mathrm{b}})) \right] \\
&= \{\text{Definition 2.15}\} \\
&\quad \beta^{\mathtt{DL}}_{\tau^{\mathrm{b}}} + (t - t_{\mathrm{b}}) \left[U\left(\tau^{\mathrm{b}}\right) - \sum_{\tau_e \in \tau^{\mathrm{b}}} csp_e(t_{\mathrm{b}}) \right] \\
&\leq \left\{ t - t_{\mathrm{b}} \geq 0 \text{ and, by Claim A.3.7, } U\left(\tau^{\mathrm{b}}\right) - \sum_{\tau_e \in \tau^{\mathrm{b}}} csp_e(t_{\mathrm{b}}) < 0 \right\} \\
&\quad \beta^{\mathtt{DL}}_{\tau^{\mathrm{b}}}.
\end{aligned}$$

This contradicts (A.6) of Claim A.3.2. This contradiction completes the proof of Lemma A.3. □

The proof of Lemma A.4 below is the same as that of the proof of Lemma 3.16 except with $\beta_{\tau'}$ replaced by $\beta^{\mathtt{DL}}_{\tau'}$ and Lemma 3.14 replaced by Lemma A.3.

▷ **Lemma A.4.** Under the patched ACS, we have

$$\forall \tau' \subseteq \tau_{\text{act}}(t) : \sum_{\tau_i \in \tau'} \sqrt{u_i} \cdot dev_i(t) \leq \beta^{\mathtt{DL}}_{\tau'}$$

for any time t. ◁

The proof of Theorem A.5 is the same as that of the proof of Theorem 3.17 except with $\beta_{\tau'}$ replaced by $\beta^{\mathtt{DL}}_{\tau'}$ and Lemma 3.16 replaced by Lemma A.4.

▷ **Theorem A.5.** Under the ACS, the response time of any task τ_i is at most

$$T_i + \frac{T_{[1]} + \frac{2m \cdot C_{[1]}}{u_{[n]}}}{2u_{[n]}} \left(2U_{\max} - u_i\right). \qquad \text{◁}$$

BIBLIOGRAPHY

Luca Abeni, Giuseppe Lipari, and Juri Lelli. Constant bandwidth server revisited. *ACM SIGBED Review*, 11 (4):19–24, 2015.

Luca Abeni, Giuseppe Lipari, Andrea Parri, and Youcheng Sun. Multicore CPU reclaiming: Parallel or sequential? In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, page 1877–1884, 2016.

Sara Afshar, Farhang Nemati, and Thomas Nolte. Resource sharing under multiprocessor semi-partitioned scheduling. In *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 290–299, 2012.

Shareef Ahmed and James H. Anderson. Tight tardiness bounds for pseudo-harmonic tasks under global-EDF-like schedulers. In *33rd Euromicro Conference on Real-Time Systems*, volume 196, pages 11:1–11:24, 2021.

Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I. Davis. An empirical survey-based study into industry practice in real-time systems. In *2020 IEEE Real-Time Systems Symposium*, pages 3–11, 2020.

Tanya Amert, Sergey Voronov, and James H. Anderson. OpenVX and real-time certification: The troublesome history. In *2019 IEEE Real-Time Systems Symposium*, pages 312–325, 2019.

James H. Anderson, Sanjoy Baruah, and Björn B. Brandenburg. Multicore operating-system support for mixed criticality. In *Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, 2009.

James H. Anderson, Jeremy P. Erickson, UmaMaheswari C. Devi, and Benjamin N. Casses. Optimal semi-partitioned scheduling in soft real-time systems. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10, 2014.

Joshua Bakita, Shareef Ahmed, Sims Hill Osborne, Stephen Tang, Jingyuan Chen, F. Donelson Smith, and James H. Anderson. Simultaneous multithreading in mixed-criticality real-time systems. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium*, pages 278–291, 2021.

Sanjoy Baruah. Feasibility analysis of preemptive real-time systems upon heterogeneous multiprocessor platforms. In *25th IEEE International Real-Time Systems Symposium*, pages 37–46, 2004.

Sanjoy Baruah and Björn B. Brandenburg. Multiprocessor feasibility analysis of recurrent task systems with specified processor affinities. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 160–169, 2013.

Sanjoy Baruah, Vincenzo Bonifaci, Renato Bruni, and Alberto Marchetti-Spaccamela. ILP-based approaches to partitioning recurrent workloads upon heterogeneous multiprocessors. In *2016 28th Euromicro Conference on Real-Time Systems*, pages 215–225, 2016.

Claude Berge. Two theorems in graph theory. *Proceedings of the National Academy of Sciences*, 43(9): 842–844, 1957.

Giorgio Buttazzo and Luca Abeni. Integrating multimedia applications in hard real-time systems. In *2013 IEEE 34th Real-Time Systems Symposium*, page 4, 1998.

Giovanni Buzzega, Gianluca Nocetti, and Manuela Montangero. Characterizing G-EDF scheduling tardiness with uniform instances on multiprocessors. In *Proceedings of the 31st International Conference on Real-Time Networks and Systems*, page 45–55, 2023.

Daniel Casini, Alessandro Biondi, and Giorgio Buttazzo. Task splitting and load balancing of dynamic real-time workloads for semi-partitioned EDF. *IEEE Transactions on Computers*, 70(12):2168–2181, 2021.

Felipe Cerqueira, Arpan Gujarati, and Björn B. Brandenburg. Linux's processor affinity API, refined: Shifting real-time tasks towards higher schedulability. In *2014 IEEE Real-Time Systems Symposium*, pages 249–259, 2014.

Micaiah Chisholm, Bryan C. Ward, Namhoon Kim, and James H. Anderson. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *2015 IEEE Real-Time Systems Symposium*, pages 305–316, 2015.

Micaiah Chisholm, Namhoon Kim, Bryan C. Ward, Nathan Otterness, James H. Anderson, and F. Donelson Smith. Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems. In *2016 IEEE Real-Time Systems Symposium*, pages 57–68, 2016.

Micaiah Chisholm, Namhoon Kim, Stephen Tang, Nathan Otterness, James H. Anderson, F. Donelson Smith, and Donald E. Porter. Supporting mode changes while providing hardware isolation in mixed-criticality multicore systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, page 58–67, 2017.

Hoon Sung Chwa, Jaebaek Seo, Jinkyu Lee, and Insik Shin. Optimal real-time scheduling on two-type heterogeneous multicore platforms. In *2015 IEEE Real-Time Systems Symposium*, pages 119–129, 2015.

Will Deacon. Asymmetric 32-bit SoCs. `https://github.com/torvalds/linux/blob/master/Documentation/arch/arm64/asymmetric-32bit.rst`, 2021. Online; accessed 20 September 2023.

Deadline Task Scheduling. Deadline task scheduling. `https://github.com/torvalds/linux/blob/master/Documentation/scheduler/sched-deadline.rst`. Online; accessed 03 June 2020.

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

Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

Paul Emberson, Roger Stafford, and Robert I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, pages 6–11, 2010.

Jeremy P. Erickson and James H. Anderson. Response time bounds for G-EDF without intra-task precedence constraints. In *Proceedings of the 15th International Conference on Principles of Distributed Systems*, page 128–142, 2011.

Jeremy P. Erickson and James H. Anderson. Fair lateness scheduling: Reducing maximum lateness in G-EDF-like scheduling. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 3–12, 2012.

Jeremy P. Erickson, UmaMaheswari C. Devi, and Sanjoy Baruah. Improved tardiness bounds for global EDF. In *2010 22nd Euromicro Conference on Real-Time Systems*, pages 14–23, 2010.

Shelby Funk, Joël Goossens, and Sanjoy Baruah. On-line scheduling on uniform multiprocessors. In *Proceedings 22nd IEEE Real-Time Systems Symposium*, pages 183–192, 2001.

Arpan Gujarati, Felipe Cerqueira, and Björn B. Brandenburg. Schedulability analysis of the linux push and pull scheduler with arbitrary processor affinities. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 69–79, 2013.

Arpan Gujarati, Felipe Cerqueira, and Björn B. Brandenburg. Multiprocessor real-time scheduling with arbitrary processor affinities: from practice to theory. *Real-Time Systems*, 51:440–483, 2014.

Hardkernel. Hardkernel linux. `https://github.com/hardkernel/linux`. Online; accessed 2 May 2024.

Godfrey H. Hardy, John E. Littlewood, and George Pólya. *Inequalities*. Cambridge University Press, 1952.

Clara Hobbs, Zelin Tong, Joshua Bakita, and James H. Anderson. Statically optimal dynamic soft real-time semi-partitioned scheduling. *Real-Time Systems*, 57(1–2):97–140, 2021.

Shinpei Kato and Nobuyuki Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. In *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 23–32, 2009.

Namhoon Kim, Micaiah Chisholm, Nathan Otterness, James H. Anderson, and F. Donelson Smith. Allowing shared libraries while supporting hardware isolation in multicore real-time systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 223–234, 2017.

Namhoon Kim, Stephen Tang, Nathan Otterness, James H. Anderson, F. Donelson Smith, and Donald E. Porter. Supporting I/O and IPC via fine-grained OS isolation for mixed-criticality real-time tasks. *Real-Time Systems*, 56(4):349–390, 2020.

Juri Lelli. taskgen. `https://github.com/jlelli/taskgen`, 2014. Online; accessed 23 Oct 2020.

Juri Lelli, Claudio Scordino, Luca Abeni, and Dario Faggioli. Deadline scheduling in the linux kernel. *Software: Practice and Experience*, 46(6):821–839, 2016.

Hennadiy Leontyev and James H. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. In *28th IEEE International Real-Time Systems Symposium*, pages 413–422, 2007.

Jiří Matoušek and Bernd Gärtner. *Understanding and Using Linear Programming*. Springer, 2007.

Rob Roy and Venkat Bommakanti. *ODROID-XU4 User Manual*. Hardkernel, 2017.

rt-app. rt-app. `https://github.com/scheduler-tools/rt-app`, 2009. Online; accessed 23 Oct 2020.

Claudio Scordino, Luca Abeni, and Juri Lelli. Energy-aware real-time scheduling in the Linux kernel. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, page 601–608, 2018.

Anand Srinivasan and James H. Anderson. Optimal rate-based scheduling on multiprocessors. *Journal of Computer and System Sciences*, 72(6):1094–1117, 2006.

Roger Stafford. Random vectors with fixed sum. https://www.mathworks.com/matlabcentral/fileexchange/9700-random-vectors-with-fixed-sum, 2024. Online; accessed 17 May 2024.

Ion Stoica and Hussein Abdel-Wahab. Earliest eligible virtual deadline first: A flexible and accurate mechanism for proportional share resource allocation. Technical report, 1995.

Stephen Tang. Identical/semi-partitioned patch. https://www.cs.unc.edu/~sytang/semipartitioned_v5.4.69.patch, a.

Stephen Tang. Uniform/semi-clustered patch. https://www.cs.unc.edu/~sytang/uscedf.patch, b.

Stephen Tang and James H. Anderson. Towards practical multiprocessor EDF with affinities. In *41st IEEE Real-Time Systems Symposium*, pages 89–101, 2020.

Stephen Tang, Sergey Voronov, and James H. Anderson. GEDF tardiness: Open problems involving uniform multiprocessors and affinity masks resolved. In *31st Euromicro Conference on Real-Time Systems*, volume 133, pages 13:1–13:21, 2019.

Stephen Tang, James H. Anderson, and Luca Abeni. On the defectiveness of SCHED_DEADLINE w.r.t. tardiness and affinities, and a partial fix. In *2021 29th International Conference on Real-Time Networks and Systems*, pages 46–56, 2021a.

Stephen Tang, Sergey Voronov, and James H. Anderson. Extending EDF for soft real-time scheduling on unrelated multiprocessors. In *2021 IEEE Real-Time Systems Symposium*, pages 253–265, 2021b.

Ismail H. Toroslu and Göktürk Üçoluk. Incremental assignment problem. *Information Sciences*, 177: 1523–1529, 2007.

Paolo Valente. Using a lag-balance property to tighten tardiness bounds for global EDF. *Real-Time Systems*, 52:486–561, 2016.

Sergey Voronov and James H. Anderson. An optimal semi-partitioned scheduler assuming arbitrary affinity masks. In *2018 IEEE Real-Time Systems Symposium*, pages 408–420, 2018.

Sergey Voronov, Stephen Tang, Tanya Amert, and James H. Anderson. AI meets real-time: Addressing real-world complexities in graph response-time analysis. In *2021 IEEE Real-Time Systems Symposium*, pages 82–96, 2021.

Bryan C. Ward, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson. Making shared caches more predictable on multicore platforms. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 157–167, 2013.

Reinhold P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

Kecheng Yang and James H. Anderson. Optimal GEDF-based schedulers that allow intra-task parallelism on heterogeneous multiprocessors. In *2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia*, pages 30–39, 2014.

Kecheng Yang and James H. Anderson. On the soft real-time optimality of global EDF on multiprocessors: From identical to uniform heterogeneous. In *2015 IEEE 21st International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10, 2015.

Kecheng Yang and James H. Anderson. On the soft real-time optimality of global EDF on uniform multiprocessors. In *2017 IEEE Real-Time Systems Symposium*, pages 319–330, 2017.

Zephyr. Zephyr. `https://github.com/zephyrproject-rtos/zephyr`. Online; accessed 21 May 2024.

Peter Zijlstra. An update on real-time scheduling on Linux. `http://archives.ecrts.org/fileadmin/files_ecrts17/ecrts17-peterz.pdf`, 2017. Online; accessed 27 June 2023.