WASHINGTON UNIVERSITY IN ST. LOUIS

McKelvey School of Engineering
Department of Computer Science & Engineering

Dissertation Examination Committee:
Jeremy Buhler, Chair
Sanjoy Baruah
James Buckley
Roger Chamberlain
Christopher Gill

Improved Models of Elastic Scheduling

by

Marion Baumli Sudvarg

A dissertation presented to
the McKelvey School of Engineering
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

May 2024
St. Louis, Missouri

Table of Contents

List of Figures

List of Algorithms

List of Tables

Acknowledgments

The work in this dissertation would not have been possible without the many mentors, colleagues, friends, and family who supported me along the way. Though it would be almost impossible to acknowledge everybody who has contributed, in one way or another, to my success, I nonetheless wish to extend my thanks to many people to whom I must attribute some piece of this work.

First, to Professor Chris Gill, who has been the best advisor I could ask for. You have taught me how to think and write like a researcher, and you have given me so much of your time, making yourself constantly available for advice and guidance these last five years. Our meetings fly by, and always end with new ideas to ponder and research directions to pursue. Thanks for remaining ever enthusiastic. Moreover, I am grateful for your flexibility and willingness to let me explore other projects and collaborate with other mentors.

Among those mentors is Professor Jeremy Buhler, who took me in for a second research rotation during my first semester, and let me stubbornly stick around to keep working on real-time GRB localization for the APT and ADAPT collaboration. This has turned into an exciting project, and that willingness was a defining factor in where I am today, and has inspired the next steps of my career. It seems like Jeremy can solve anything, and his sharp ability to communicate results makes him a valuable resource when stuck on thinking about a problem or writing about the solution. Thank you for chairing my dissertation committee and for giving me so much of your time.

Thanks also to Professor Roger Chamberlain, who has gotten me more involved in the digital hardware design for ADAPT. I started with no experience in this area, and through his help (and help from his PhD student Chenfeng Zhao, to whom I also extend my gratitude) I have had the opportunity to learn about high-level synthesis of digital accelerator logic. I also want to thank Roger for funding my work, and for paying (from his grants) for my travels to several conferences. Roger's door is always open, and he never fails to give encouragement and advice when needed.

To Professor Jim Buckley, I should start by thanking you for taking me on as a part-time summer research assistant when I was an undergraduate student. I had some lingering regrets about not spending more time in your lab in those days, and so I am overjoyed that you have let me be involved in the development of ADAPT. I am looking forward to the next year, as we continue that development, and branch out to broader problems of coordinated real-time localization and follow-up observations in time-domain and multi-messenger astrophysics. I am fascinated and inspired by your work, and I hope to someday gain a fraction of your knowledge about the mysteries of our universe. Thank you also for joining my dissertation committee as the outside member.

And to my final committee member, Professor Sanjoy Baruah, I am always in awe of your vast knowledge and keen insights into scheduling theory. You always manage to bring clarity to complicated problems, distilling them into straightforward models and solutions. For somebody with as many projects and collaborations as you have, I always appreciate that you are willing to give me so much of your time. I also aspire to learn from your humility, tact, and grace in communicating with other people.

I also want to thank Professor Ning Zhang, who is an incredible systems researcher, and always asks the difficult questions to make sure we motivate and position our work correctly. I have learned a lot from you, and I am grateful that you have been willing to collaborate with me.

I extend my thanks to Rick Gray, an earlier mentor of mine. Rick took me under his wing when I started as a network administrator at Seiler Instrument, and later trusted me enough to promote me to information systems manager. He taught me a great deal about solving systems problems, and a great deal more about professionalism in the workplace. I think that the sliver of his work ethic that made its way to me has helped me stay focused along the way.

To my fellow students, Ye Htet, Ao Li, and Daisy Wang: thank you for being great collaborators. You have all helped immensely with my work, and without your implementation and experimental efforts, this dissertation would not be where it is today. Ye, you have always had clever ideas for analyzing and improving the GRB localization pipeline, and I look forward to seeing where machine learning can take us. Ao, you are one of the most hardworking and dedicated people I know, and you push me to new heights. Your insights into building and framing new systems never fail to impress me. And Daisy, your work has progressed so much in such a short time. I am in awe of what you have accomplished with FIMS, and I look forward to working together more in the future.

I would also like to thank my parents, who taught me the value of education from an early age. When I was a child, they took a lot of time from their busy lives to read to me, to give me math problems to do, and to encourage me to write. Writing and problem solving are now joys of mine, rather than chores.

Finally, to my wife Joyce, I extend all of my love and thanks. Thank you for being there for me since I started the PhD, through all of the disappointments, stressful times, and late nights. But also for all of the achievements and opportunities to journey to conferences; you have been my best friend and traveling companion, and I look forward to a lifetime of many more trips together.

My work in this dissertation has been supported directly by NSF grants CSR-1814739, CNS-1763503, CNS-2141256, CPS-2229290; NASA grant 80NSSC21K1741; and a gift from BECS Technology, Inc. Other support for work in this dissertation has been provided by NSF grants CNS-2038995, CNS-2238635; a Washington University CSE/EECE seed grant; and Swedish Research Council grant 2018-04446.

Marion Baumli Sudvarg

Washington University in St. Louis
May 2024

ABSTRACT OF THE DISSERTATION

Improved Models of Elastic Scheduling

by

Marion Baumli Sudvarg

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2024

Professor Jeremy Buhler, Chair

In real-time computing systems, *timely* execution is a requirement of *correct* execution. Such systems are widely found in robotics and autonomous vehicle applications, mobile spectrometry of atmospheric aerosols, real-time hybrid simulation for natural hazards engineering, and in prompt localization of transients such as gamma-ray bursts for time-domain and multi-messenger astrophysics. *Elastic scheduling* provides a framework to adjust computational rates and workloads in systems for which timeliness cannot otherwise be guaranteed. While originally proposed for periodic tasks executing on a single processor, elastic scheduling has since been extended to sequential and parallel execution on multiple processors and to earliest deadline first scheduling of constrained-deadline tasks.

This dissertation expands the state of the art in elastic scheduling in three key directions. First, it proposes and analyzes new algorithms for elastic scheduling with provably better execution time complexity compared to the prior work, enabling better guarantees of timeliness associated with adapting to new execution states. Second, it presents new extensions of the elastic model to other common scheduling paradigms, including fixed-priority scheduling of constrained-deadline tasks and task systems for which execution semantics demand that periods take harmonic values. Third, it considers the impact of adaptation on system performance as a whole, and demonstrates how changes in task periods and workloads can be realized to optimize expected system outcomes within the constraints of schedulability.

In doing so, this dissertation paves the way toward a richer set of adaptive scheduling and execution models in simulation, localization, and control applications.

Chapter 1

Introduction

1.1 Overview

An increasingly connected world has driven rising demand for applications that respond to events (e.g., user requests, a changing environment, or transient astrophysical phenomena) in *real time*. Such applications are deployed on diverse system platforms, from those highly constrained in size, weight, and power (SWaP) such as microrobots [139, 1, 103] and satellites [30, 144]; to distributed systems such as those found in autonomous vehicles [82] and avionics [65, 67, 57]; and even to large computing clusters and edge cloud infrastructure [164, 165].

Real-time systems must remain adaptable to the diversity of environments and platforms in which they operate. For some applications, exogenous conditions drive dynamic workloads and deadlines [68]. For example, real-time aerosol sampling, which may allow adaptive flight patterns for atmospheric survey missions, has workloads that vary with the density of the sampled particles, the vibrations of the enclosing chamber, and the changing reflectivity of the chamber walls [169]. Gamma-ray burst (GRB) localization for prompt alerts to follow-up instruments has a workload and deadline informed by the GRB's brightness, time profile, and spectral-energy distribution, none of which are known a priori [145]. Other applications are built to be resilient in the face of component failure. For example, on a space-based computational platform, radiation damage might render a processor core inoperable. Nonetheless, the system as a whole must remain operational, even in a degraded state.

Elastic real-time scheduling models provide a framework for dynamic task adaptation to guarantee timely computation even on a system that becomes overloaded due to changes in computational demand, temporal requirements, or resource availability. While prior work has extended elastic adaptation to several scheduling models, key questions remain to be answered. ***This dissertation expands the state of the art in elastic scheduling*** in three primary directions:

1. It proposes and analyzes new algorithms for elastic scheduling with provably better execution time complexity compared to those in prior work.
2. It presents new extensions of the elastic model to common scheduling paradigms.
3. It considers the impact of adaptation on system performance as a whole, and demonstrates how elastic scheduling models can be constructed to optimize expected system outcomes within the constraints of schedulability.

1.2 Background and Context

1.2.1 Real-Time Systems Scheduling

Real-time systems are computing systems for which *temporal* and *functional* correctness are equally important. In these systems, *timely* execution is a requirement of *correct* execution; therefore, *scheduling* computations to guarantee timeliness is a first-class concern. For over half a century, many such systems have been modeled using Liu and Layland's recurrent task model [97]. Under this model, a *task* is defined as a unit of repeated execution, e.g., the reading of a sensor; the execution of a single time step in an earthquake simulation [63, 64]; or as we have discussed in prior work, the reconstruction of a single Compton-scattered gamma ray from a GRB [30, 48, 144, 153, 171, 145, 147, 75, 47, 146, 76, 154]. Tasks are characterized by:

- A worst-case execution time (WCET) value representing an upper-bound on the time to complete the computation of a single instance of the task, if uninterrupted on a single processor core. We note that this is platform-dependent: a task's WCET depends on the processor on which it executes.
- A period, representing the time between releases of repeated instances of the task. Under "sporadic" task models, the period represents the *minimum* time between releases.

- A relative deadline, representing the time, from its release, by which a single instance of a task must complete. If an instance of a task must complete only before release of the next instance, then the relative deadline is set equal to the period duration, and the task is referred to as "implicit-deadline". However, if the deadline takes an earlier value, we call it a "constrained-deadline" task (the relative deadline is constrained to not exceed the period duration).

Uniprocessor Scheduling

A task's utilization is defined as its worst-case execution time divided by its period; intuitively, this is the largest fraction of a single processor's time that it must use. For implicit-deadline tasks, Liu and Layland [97] demonstrated that with preemptive fixed-priority rate-monotonic (RM) and earliest-deadline first (EDF) scheduling, a set of tasks are schedulable on a single processor if their total utilization does not exceed a given bound.

Preemptive RM and EDF scheduling have thus become common paradigms when implementing schedulers and analyzing schedulability. Under RM, each *task* is assigned a priority according to its period; all jobs of a task with a larger period have a lower priority. Under EDF, each instance, or *job*, of a task is assigned a priority according to its absolute deadline; a job that must complete earlier is given a higher priority. When a job is invoked, if it has a higher priority than the currently executing job, then it will begin to execute immediately; the lower-priority job is said to be *preempted*. For sets of constrained-deadline tasks, more computationally-complex analysis is needed to verify schedulability.

Federated Scheduling

The increasing computational demand of many real-time applications motivates scheduling strategies for task sets with *intra-task parallelism*. Systems hosting multiple tasks, where individual tasks may need to execute on more than one processor to meet their deadlines, are becoming more common. Such systems can be increasingly found in autonomous vehicles [82], computer vision systems [60], real-time hybrid testing environments [63, 64], and, as we have shown, in satellite telescopes [144, 153, 171, 75, 146]. Deciding how to schedule such tasks, i.e., how and when to allocate computational resources such that all execution completes before its corresponding deadline, is a fundamental challenge for these systems.

Federated scheduling [94] — which has been used in real-world applications such as real-time hybrid simulation (RTHS) [121] — proposes one approach to this problem: high-utilization tasks (i.e., those with utilization exceeding 1) are each assigned a dedicated set of cores on which they alone execute; low-utilization tasks are then scheduled on the remaining cores using a multiprocessor scheduling algorithm for sequential tasks, e.g., partitioned EDF [12], global EDF [52, 93, 5], or partitioned RM [46].

In this model, parallel tasks are represented as a set of *subtasks*, each of which must execute sequentially, i.e., an individual subtask cannot execute on more than one processor at once. However, multiple subtasks may execute in parallel with one another, provided that precedence constraints are satisfied: a subtask is said to *precede* another if it must complete its execution before the other begins. If subtask A precedes subtask B, we also say that B *succeeds* A. The partial ordering induced by the precedence relation over the set of subtasks naturally gives rise to a directed acyclic graph (DAG) with vertices corresponding to subtasks. A directed edge connects vertex A to B if and only if the corresponding subtask A directly precedes subtask B, i.e., there is no other subtask that precedes B and succeeds A.

Each subtask is characterized by its individual worst-case execution time. The workload of the parallel task is defined as the sum of its subtasks' WCETs, i.e., the upper bound of its execution time on a single processor core. A parallel task is also characterized by its *span*, or the total workload of the subtasks along the critical path of its DAG, where each vertex is weighted by the corresponding subtask's WCET. In other words, the span represents the upper bound on time to complete the task if given an infinite number of processors on which to execute. For a parallel task characterized in this way, it is known to be NP-hard to determine the *minimum* number of processor cores necessary for it to complete by its deadline [159]. However, in [94], the authors demonstrate that a *sufficient* number of cores can be computed in constant time given a parallel task's workload, span, and deadline.

1.2.2 Elastic Scheduling

The scheduling strategies and their corresponding analysis discussed above simply state whether a set of tasks is *schedulable* (i.e., whether the scheduling strategy guarantees that every instance of every task will meet its deadline) or *unschedulable*. A system for which resources are insufficient to guarantee schedulability is said to be in an *overloaded* state. Traditionally, sets of recurrent workloads operate under a rigid model where their parameters are assumed to be fixed. If they are deemed schedulable, then the system may execute as normal, but overload is to be avoided.

However, in many systems, the paradigm of defining timing constraints as static task properties may be relaxed. To this end, elastic real-time scheduling models provide a framework for dynamic adaptation of task utilizations in response to system overload (e.g., during online mode changes or for admission of new tasks). The original model proposed by Buttazzo et al. [39, 40] considers uniprocessor scheduling of implicit-deadline tasks. Using a physical analogy, it represents each task's utilization as a spring. The task's "elasticity" reflects its flexibility to adapt its utilization to a lower quality of service; for example, a more important task might be considered less elastic. The total length of the springs, placed end-to-end, represents system utilization. If this exceeds the schedulable bound, a compressive force is applied to the system. Each spring (and corresponding utilization) is compressed proportionally to its elasticity until the total utilization no longer exceeds the bound or until the task reaches its minimum serviceable utilization; task periods are adjusted accordingly.

Elastic scheduling has since been extended to multiprocessors [118], to EDF scheduling of constrained-deadline tasks [44, 45, 13], and to federated scheduling of parallel real-time tasks for which periods [120] and workloads [119] are compressed over continuous ranges[1], as well as tasks for which only a discrete set of candidate utilizations may be accommodated [121].

1.3 Contributions

This dissertation makes the following contributions to elastic scheduling of real-time systems.

[1]We refer to tasks for which utilization compression is realized by extending periods as *rate-elastic* or *period-elastic*. If task workloads are reduced such that WCET decreases, we instead call them *computationally-elastic* or *workload-elastic*.

1.3.1 Extensions to New Scheduling Models

Fixed-Priority Scheduling of Constrained-Deadline Tasks

Buttazzo's elastic scheduling model in [39, 40] was originally formulated for implicit-deadline tasks for which schedulability may be analyzed using the simple utilization-bound conditions of Liu and Layland [97]. Under this model, task utilizations are simply decreased proportionally to their "elasticity" until the total utilization no longer exceeds the bound.

For constrained-deadline tasks, the analysis is more complex — indeed, for RM scheduling of constrained-deadline tasks, determining schedulability is NP-complete [58], and for EDF scheduling, it is coNP-complete [59]. Nonetheless, the semantics of implicit deadlines do not capture the temporal requirements of many systems. Consider a control application where plant state is sensed at the instant a job is released, and a control signal is then computed to be applied to the plant at some set time in the future. In [13], the authors consider the control loop as an elastic task where increasing the period allows the control loop to be executed less frequently, while nevertheless applying the control signal at the intended instant relative to the sampled state. In this case, the task's deadline remains constant while the period increases, making it a constrained-deadline elastic task.

While elastic models have been applied to EDF scheduling of constrained-deadline tasks in [44, 45, 13], it has not yet been extended to constrained-deadline fixed-priority scheduling. While EDF is an optimal scheduling algorithm under many models, including for preemptive uniprocessors [77], it is not implemented in many operating systems; for example, the seL4 microkernel provides only task-level fixed-priority scheduling [102], and until version 3.14, EDF was not available in the Linux kernel [51]. Furthermore, many priority-based shared resource access protocols, such as the priority ceiling [135] and immediate priority ceiling [9, 38] protocols, assume task-level fixed priorities. To this end, ***this dissertation extends elastic scheduling to fixed-priority constrained-deadline tasks.***

Harmonic Periods

Many control systems, such as those found in robotics applications [92] and RTHS for structural integrity [121], demand harmonic rates among their constituent tasks. In applications that capture and process frames from multiple sensing devices to be aggregated in backend processing tasks, such as simultaneous localization and mapping (SLAM) [88] and real-time mobile spectrometry [167], harmonic task periods guarantee consistent temporal alignment. Furthermore, task sets with harmonic periods have hyperperiods equal to the largest period [29, 130], which reduces the size of scheduling tables in time-triggered systems [83] and constrains the test set in processor demand analysis [20].

Selecting harmonic periods from within acceptable intervals is nontrivial. Nasri et al. [112] formalized the problem, and proposed an approach to solve it in time linear in the number of tasks for restricted cases. In general, though, the algorithm's running time "can exponentially grow." In [111], Nasri and Fohler identify another restriction of the problem that can be solved in polynomial time, but they provide "no guarantee for reasonable computational complexity" in general.

This dissertation extends elastic scheduling to sets of tasks with harmonic period constraints. It is the first to reason about the problem of assigning task periods from continuous intervals such that *(i)* periods remain harmonic, while *(ii)* respecting the relative flexibility of each task to adapt its utilization, but still *(iii)* guaranteeing the schedulability of every task.

Federated Scheduling with Low-Utilization Tasks

Prior extensions of elastic models to federated scheduling of parallel tasks in [120, 119, 121] consider high-utilization parallel tasks that must execute on dedicated cores. Under the original federated scheduling model in [94], low-utilization tasks (those that individually require only a single core) are scheduled concurrently on any remaining cores not allocated to the high-utilization tasks. Therefore, to fully extend elastic scheduling to the federated paradigm, a model must consider the allocation of computational resources to both low- and high-utilization tasks.

However, the approaches in the prior work assume a *fixed allocation of cores for high-utilization parallel tasks*. These approaches do not consider that compression of utilization over *all* tasks in the system may change the number of cores required by low-utilization tasks. ***This dissertation extends the elastic frameworks for federated scheduling to dynamically allocate cores during joint compression of both low- and high-utilization tasks.***

1.3.2 Elasticity to Optimize System Outcomes

One advantage of Buttazzo's elastic scheduling model is that the policy for assigning task utilizations "is implicitly encoded in the elastic coefficients provided by the user (for example, based on task importance)" [40]. In some applications, importance may be qualitative; nonetheless, elastic coefficients are quantitative values, and the choice of values may determine system outcomes. In this work, we address questions about what the elastic constant represents, and explore what it means to apply elastic compression to individual tasks or even subtasks. Furthermore, we explore scenarios where elastic coefficients are insufficient to capture the joint impact of task compression on system performance as a whole.

Selecting Elastic Constants

In [44, 45], it was shown that utilizations satisfying the elastic scheduling semantics of proportional compression could be obtained by solving a quadratic optimization problem. By mapping this problem onto the original elastic model, ***this dissertation demonstrates how to assign elastic constants to reflect the first-order impact on system performance loss associated with reducing the rate of each task.*** We demonstrate this in the context of two applications: real-time aerosol monitoring in the fast integrated mobility spectrometer (FIMS) [166] and simultaneous localization and mapping (SLAM) for mobile robots and drones [42].

Subtask-Level Elastic Scheduling

Prior work on computationally-elastic parallel tasks [119] only considers the aggregate reduction of a parallel task's overall workload, and not the individual implications of reducing the workloads of each *subtask*. Changing the computational workload of each subtask may fundamentally affect quality of outcome (e.g., control performance, prediction accuracy, etc.)

in different ways; this has been discussed in the context of autonomous vehicles [7] and we have shown this to be true for GRB localization [146].

Furthermore, individual changes in subtask workloads may affect the task's *span*, or critical path length. The elastic model in [119] determines schedulability according to the approach in [94] for federated scheduling, in which cores are allocated according to the workload, span, and deadline parameters of each task. As it decreases task workloads, the model in [119] holds the span constant. However, as a task's workload decreases, its critical path length may also decrease as the workloads of the subtasks (or even the set of subtasks) along the path changes, which further reduces the necessary allocation of processor cores to the task. For this effect to be captured, an elastic model must be cognizant of the DAG structure induced by the precedence constraints among the subtasks composing each parallel task.

To this end, ***this dissertation proposes a new model of subtask-level elasticity for federated scheduling of parallel tasks.*** Under this model, each subtask is assigned a range of acceptable workloads and its own elastic constant. Elastic compression is thus applied to the complete collection of both parallel task subtasks *and* sequential tasks in the system to guarantee schedulability according to the resulting execution times and spans.

Utility-Driven Parameterized Compression

The existing semantics of elastic scheduling models may face important limitations when applied to real-world task systems. Elastic compression scales the utilizations of tasks proportionally to their elastic coefficients. This coefficient is meant to represent a task's importance or relative adaptability. As we will show via problem transformation, this implies mathematically that the coefficient captures a first-order quadratic relationship between each task's utility (or subtask's workload, in the case of subtask-level elastic scheduling) and the quality or utility of system outcome.

However, these semantics might not capture second-order effects and nonlinearities present in the relationships between computational adaptation of tasks and subtasks and the results they produce. For example, we have developed a real-time GRB localization pipeline to enable prompt multi-messenger follow-up observations of transient bursts [144, 145]. The pipeline consists of several stages, including event reconstruction, approximation of an initial source direction, and subsequent iterative refinement of the estimate. Compressing the workload of the event reconstruction stage by reconstructing fewer events also decreases the workload of each refinement iteration; the two subtasks cannot be considered independently of each other [145, 146]. We therefore argue that elastic scheduling should instead be reframed in the context of a richer family of optimization problems over multiple parameterized degrees of freedom for which a task's workload or period can be adjusted within the constraints of schedulability. Toward this vision, ***this dissertation proposes an approach for utility-driven parameterized workload adaptation of parallel tasks.***

1.3.3 Improved Execution Time Complexity

Elastic scheduling algorithms realize compression of task utilizations by selecting, for each task, a period or workload assignment. While elastic scheduling models are useful for adjusting a predefined set of tasks to be scheduled on a resource-constrained system, Buttazzo's original model was primarily intended for use in dynamic and open systems where the set of active tasks may change [39, 40]. Therefore, while it is always desirable to obtain efficient algorithms, it is especially important for elastic scheduling algorithms that adapt in response to *online* changes to have provable execution time bounds. In this dissertation, we present several approaches that use more computationally-expensive *offline* characterization and setup steps to construct data structures that enable polynomial or pseudo-polynomial time online adaptation.

Implicit-Deadline Tasks

The original algorithms in [39, 40] for elastic scheduling of implicit-deadline tasks on a uniprocessor execute in time quadratic on the number of tasks. ***This dissertation presents an algorithm that executes in quasilinear time on the number of tasks, and in linear time when a new task arrives or system parameters change.***

Furthermore algorithms for elastic scheduling of implicit-deadline sequential tasks (i.e., tasks that *individually* require no more than a single core to execute) on multiple processors have been proposed in [118]. We extend our improved algorithm to two of the multiprocessor scheduling paradigms discussed in that work: *fluid* scheduling and *partitioned EDF.* Fluid scheduling is an abstraction under which each individual tasks are assigned a fraction of a processor at each instance in time [19]. Partitioned EDF, which is more practical for implementation [41, 121] assigns each task to a single processor, then schedules each processor independently according to EDF [12].

The prior extension of elasticity to fluid scheduling in [121] executes in time quadratic on the number of tasks, while its algorithm for partitioned EDF is pseudo-polynomial in the number of tasks, number of processors, and a selected parameter to tune the precision of the result. On the other hand, ***this dissertation extends its quasilinear and linear-time algorithms to both fluid and partitioned EDF scheduling*** while also making other improvements to the original algorithm for partitioned EDF elastic scheduling.

Harmonic Periods

As previously mentioned, selecting harmonic periods from within acceptable intervals is nontrivial; indeed, Nasri and Fohler argue that the complexity of the problem grows exponentially with the number of tasks [112]. However, ***this dissertation demonstrates a pseudo-polynomial algorithm for assigning harmonic periods to tasks***, though it proves that the problem likely cannot be solved in polynomial time.

Furthermore, while we show that the problem of elastic scheduling with harmonic periods is NP-hard, we consider a natural restriction in many systems that enables efficient online adjustment. In many multi-time stepping (MTS) applications such as decomposition of RTHS [31] for natural hazards engineering, and in applications such as ORB-SLAM [110] where front-end data collection tasks capture and process frames from sensing devices which must be aggregated downstream, task periods must respect an a priori total ordering. Under this restriction, ***this dissertation shows that a lookup table, generated offline, enables polynomial-time adjustment of task periods in response to changes in available utilization.***

Parameterized Adaptation for Dynamic Workloads and Deadlines

Many real-time systems run in dynamic environments where exogenous factors inform task workloads and deadlines, which may not be known prior to job release. A job of a task that would otherwise miss its deadline may adapt to remain schedulable by executing in a degraded state that reduces its workload. Under richer models of adaptation over multiple parameterized degrees of freedom, immediately finding the optimal execution mode to guarantee timely completion may be difficult.

To this end, we demonstrate an offline approach for identifying the parameterized degrees of freedom over which a task's workload can be adjusted, then characterize the impact of workload reduction on response time and utility. This enables construction of a Pareto-optimal surface over which efficient search, interpolation, and extrapolation enable online selection of task parameters at time of job release. In doing so, ***this dissertation provides an approach to bounded-time online utility-driven workload adaptation.***

1.4 Organization

The remainder of this dissertation is structured as follows.

Chapter 2 proposes an algorithm for uniprocessor elastic scheduling of implicit-deadline task systems that runs in time quasilinear on the number of tasks. It adapts the algorithm for linear-time compression in response to admission of a new task or a change in available utilization. It demonstrates, both through analysis of the algorithms and evaluations of synthetic task sets, the execution time improvements that may be realized. Portions of the chapter were published as "Linear Time Admission Control for Elastic Scheduling" in the Springer Real-Time Systems journal [150].

Chapter 3 shows how to extend the algorithm in Chapter 2 to multiprocessor scheduling of sequential tasks in the context of both fluid scheduling and partitioned EDF. It also proposes more efficient alternative implementations of the prior elastic algorithm for partitioned EDF scheduling. It compares these implementations to the new algorithm, and evaluates the tradeoffs between task schedulability and execution time efficiency.

Chapter 4 extends the elastic model to fixed-priority scheduling of constrained-deadline tasks on a uniprocessor. It proposes and analyzes four algorithms toward solving the problem, compares their performance through empirical evaluation, and suggests different scenarios where each one may be applicable. Portions of the chapter were published as "Elastic

Scheduling for Fixed-Priority Constrained Deadline Tasks" at ISORC 2023, winning the best paper award [143].

Chapter 5 extends elastic scheduling to tasks with periods constrained to harmonic values. It demonstrates that the problem of harmonic period selection from specified intervals can be solved in pseudo-polynomial time, but likely not in polynomial time. It also demonstrates that an application of elastic scheduling under these constraints is NP-hard. However, under the restriction that an a priori total order is applied to task periods, it presents a polynomial-time algorithm to adapt in response to changes in available utilization. This chapter evaluates its algorithms in the context of synthetic tasks and two real-world applications: ORB-SLAM3 [42] and FIMS [166]. It discusses remaining limitations of these models and suggests directions for future work. Portions of the chapter were published as "Elastic Scheduling for Harmonic Task Systems" at RTAS 2024 [151].

Chapter 6 presents our model for subtask-level elasticity under federated scheduling. It formulates a quadratic optimization problem to assign subtask workloads under the model, and demonstrates two approaches to constructing it as a mixed-integer quadratic program (MIQP) that can be solved by off-the-shelf tools. It also proposes a more efficient dynamic-programming (DP) technique that can realize compression online with pseudo-polynomial time complexity. It also shows how the DP-based approach can be applied to joint compression of low-utilization tasks. Finally, it evaluates the proposed techniques on large numbers of synthetic task sets.

Chapter 7 presents our approach for workload adaptation over multiple parameterized degrees of freedom to optimize utility within the bounds of schedulability. It discusses techniques for online adaptation of highly-parallel fork-join tasks when workloads and deadlines are not known prior to job release. It demonstrates and evaluates this approach in the context of our GRB localization pipeline [144, 145], and presents additional optimizations such as slack reclamation. Finally, it considers possible directions toward considering this approach as a more general collection of frameworks in which elasticity plays a part. Portions of the chapter were published as "Parameterized Workload Adaptation for Fork-Join Tasks with Dynamic Workloads and Deadlines" at RTCSA 2023 [146].

Chapter 8 concludes this dissertation. It discusses related work, placing our elastic models in a broader context, and proposes many directions that this research may take in the future.

Chapter 2

An Efficient Algorithm for Uniprocessor Implicit-Deadline Tasks

Portions of this chapter were published as "Linear Time Admission Control for Elastic Scheduling" in the Springer Real-Time Systems journal [150].

2.1 Introduction

Existing work on elastic scheduling considers algorithms that compress (i.e., reduce) the utilizations of tasks to guarantee schedulability on an otherwise overloaded system. While such models are therefore useful for adjusting a predefined set of tasks for execution on a resource-constrained system, Buttazzo's original elastic scheduling model was primarily intended to enable online adaptation in dynamic and open systems [39, 40]. Thus, it is important for elastic scheduling algorithms to be both *efficient* and provide *bounded-time* complexity guarantees.

Under Buttazzo's model, each task is assigned a continuous range of utilizations at which it may execute and a constant elasticity parameter that "specifies the flexibility of the task to vary its utilization" [39]. For implicit-deadline tasks on a uniprocessor, utilization-based schedulability analysis deems a set of tasks schedulable if their total utilization does not exceed some bound. If a system is overloaded (i.e., if the total maximum utilization demanded by all tasks exceeds the schedulable bound) then each task's utilization is compressed proportionally to its elasticity until total utilization reaches the bound. Simple arithmetic over each task's parameters determines the amount of compression necessary. *However*, if this would reduce some task's utilization below its minimum, then 1 that task's utilization is fixed to its minimum value, ② the task is removed from the set of elastic tasks, and ③ the compression algorithm is run again over the remaining tasks. Because this procedure may repeat as each task reaches its minimum utilization, the total execution time is quadratic in the number of tasks.[2]

In this chapter, we present a modified algorithm for elastic scheduling of implicit-deadline tasks on a uniprocessor that provides better execution time complexity. The key observation is as follows. Given a task's range of allowed utilizations and elasticity parameter, we can characterize the maximum "amount" of compression that can be applied before it reaches its minimum utilization constraint. By sorting tasks (which takes quasilinear time) in order of how quickly they will reach this constraint, any compression (e.g., when a new task arrives, or when available utilization changes) can be achieved in linear time over the set of tasks. We demonstrate that ***this allows for faster online adaptation compared to Buttazzo's algorithm.***

The remainder of this chapter is organized as follows:

- Section 2.2 develops the system model used in this chapter and provides necessary background on uniprocessor elastic scheduling of implicit-deadline tasks.
- Section 2.3 presents our improved algorithm and proves its correctness.
- Section 2.4 empirically compares our proposed algorithm to Buttazzo's.
- Section 2.5 concludes the chapter.

2.2 Background and System Model

This section provides necessary background on real-time scheduling before introducing Buttazzo's elastic scheduling model from [39, 40].

[2]When presenting the algorithm in [39], Buttazzo et al. suggest that it runs in quadratic time only when tasks have minimum utilization constraints, and that it is otherwise linear. We note, on the contrary, that tasks *must* be assigned a minimum utilization of at least 0; otherwise, the algorithm might assign negative utilizations. We illustrate this in Example 1. Therefore, the algorithm cannot be guaranteed to have better than quadratic time complexity.

2.2.1 Uniprocessor Scheduling of Implicit-Deadline Tasks

Liu and Layland's recurrent task model has been a dominant scheduling paradigm in real-time systems for over half a century [97]. Under their model, a system executes a set $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$ of n tasks. Each task τ_i is a unit of repeated execution, e.g., the reading of a sensor; the execution of a single time step in an earthquake simulation [63, 64]; or as we have discussed in prior work, the reconstruction of a single Compton-scattered gamma ray from a GRB [30, 48, 144, 153, 171, 145, 147, 75, 47, 146, 76, 154]. Each repeated instance of a task is referred to as a job; $J_{i,j}$ denotes the j^{th} job of task τ_i. A job is called *active* if it has been released, but has not yet completed execution. Each task τ_i is characterized by the following parameters, which are illustrated in Figure 2.1.

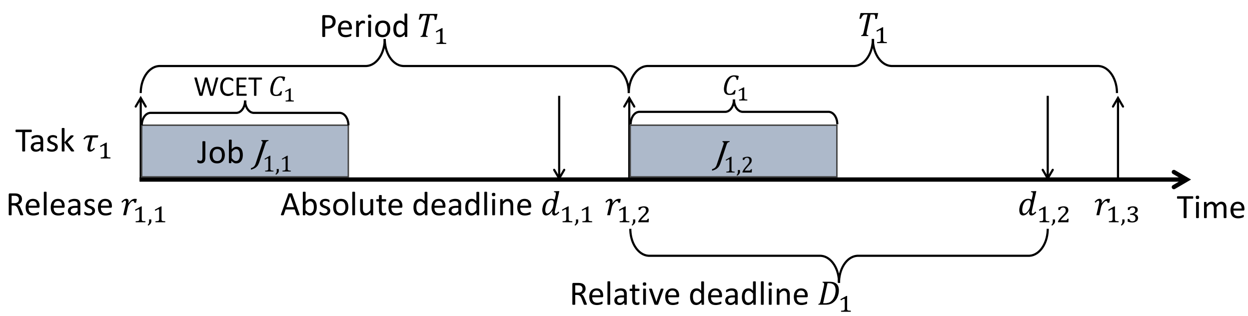


Figure 2.1: An illustration of a task under the Liu and Layland [97] model. This is a *constrained-deadline* — but not an *implicit-deadline* — task, since $D_i < T_i$.

- C_i: the worst-case execution time (WCET), which represents an upper-bound on the time to complete a single uninterrupted job on one processor core. (This is platform-dependent, since a task's WCET depends on the processor on which it executes.) We may also refer to the WCET as the task's *workload*.

- T_i: the period, representing the time between jobs. If a job $J_{i,j}$ becomes available for execution at time $r_{i,j}$, we call this the *release* time of the job. For periodic tasks, the release times of subsequent jobs are separated by exactly T_i time units, i.e., $r_{i,j} - r_{i,j-1} = T_i$. For sporadic tasks, the period represents the minimum inter-arrival time of jobs, i.e., $r_{i,j} - r_{i,j-1} \geq T_i$. Sporadic tasks are commonly event-driven, in which case the period represents the minimum time between event arrivals. For Poissonian events without a minimum interarrival time, a period may nonetheless be enforced explicitly (e.g., via interrupt masking [129]) or implicitly (e.g., due to dead times in sensor readout electronics [154]).

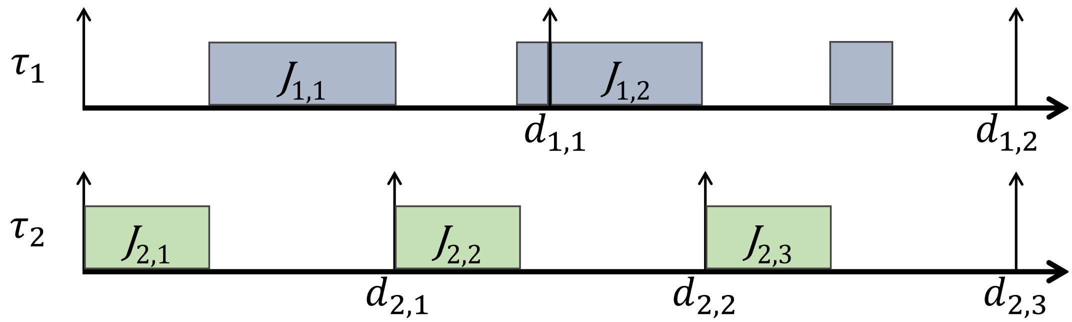
- D_i: the relative deadline, representing the elapsed time from its release by which any single job of the task must complete. If job $J_{i,j}$ is released at time $r_{i,j}$, then we say its *absolute* deadline $d_{i,j}$ is $r_{i,j} + D_i$. For implicit-deadline tasks, $D_i = T_i$, i.e., every job of a task must complete before the next job of the same task is released. Constrained-deadline tasks are those for which the deadline may be shorter than the period, i.e., $D_i \leq T_i$. We may refer to the relative deadline D_i more simply as the "deadline," i.e., if we do not specify whether we are referring to the *relative* or *absolute* deadline of the task, the reader may assume the relative deadline by default.

From these parameters, we can also derive the utilization $U_i = C_i/T_i$ of a task τ_i. Intuitively, this is the largest fraction of a single processor's time that the task must use.

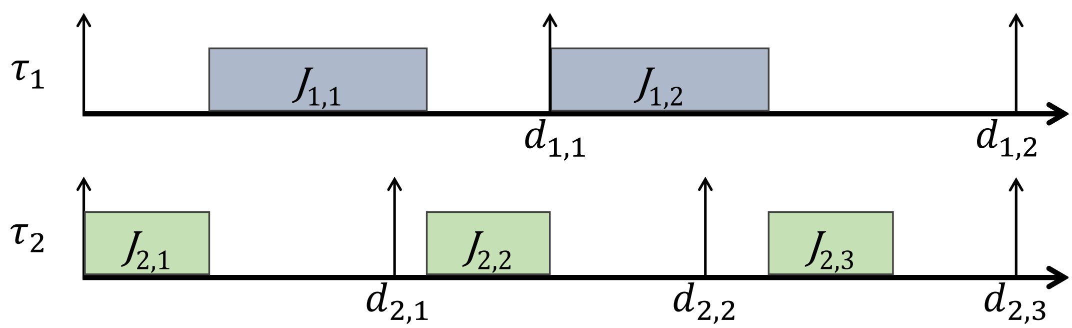
For a collection of active jobs, a scheduling algorithm determines, at every instant in time, which job should execute and on which processor. In this chapter, we consider two scheduling approaches for execution on a single processor: preemptive fixed-priority (FP) and preemptive earliest-deadline first (EDF).

Under task-level fixed priority scheduling, each *task* is assigned a priority according to its period. For implicit-deadline tasks, rate-monotonic (RM) scheduling — where all jobs of a task with a shorter period are assigned a higher priority than those of a task with a longer period — is an optimal fixed-priority algorithm. Under EDF, on the other hand, each individual *job* is assigned a priority according to its absolute deadline; an earlier absolute deadline corresponds to a higher priority. EDF is optimal for implicit-deadline tasks. Under both algorithms, if a job is released with a higher priority than the currently-executing job, then it will preempt the execution and be serviced immediately. These concepts are illustrated in Figure 2.2.

For uniprocessor scheduling of implicit-deadline tasks, Liu and Layland [97] demonstrated that a set Γ of n tasks τ_i are schedulable under RM or EDF if their total utilization $U_{\text{SUM}} = \sum_i U_i$ does not exceed a given bound. For EDF, that bound is 1. For RM, it is $n(2^{1/n} - 1)$, or 1 if periods are harmonic [84].



(a) An example of RM scheduling. Because τ_1 has a longer period than τ_2, its jobs are always scheduled at a lower priority. Jobs $J_{1,1}$ and $J_{2,1}$ are both released at time 0, so $J_{2,1}$ executes until it completes at time 4. $J_{1,1}$ is then able to execute for 6 time units until the release of $J_{2,2}$ at time 10. $J_{2,2}$ executes until it completes at time 14, whereupon $J_{1,1}$ can execute for its remaining 1 time unit. At time 15, job $J_{1,2}$ is released, and it executes until time 20 when it is preempted by the release of job $J_{2,3}$. Job $J_{2,3}$ executes until completion at time 24, whereupon job $J_{1,2}$ may execute for its final 2 time units, completing at time 26.



(b) An example of EDF scheduling. Unlike RM, or other task-level fixed priority scheduling algorithms, individual *jobs* are prioritized according to their absolute deadlines. When jobs $J_{1,1}$ and $J_{2,1}$ are released simultaneously at time 0, $J_{2,1}$ is prioritized because its absolute deadline at time 10 is earlier than that of $J_{1,2}$ at time 15. $J_{2,1}$ executes for its complete 4 time units, whereupon $J_{1,1}$ begins to execute. At time 10 when $J_{2,2}$ is released with an absolute deadline of 20, it does *not* preempt $J_{1,1}$, which is able to run until it completes at time 11. At this point, $J_{2,2}$ begins execution and runs, uninterrupted, until time 15. $J_{1,2}$ is released at time 15 and begins to execute. It is still executing at time 20 when $J_{2,3}$ is released. Because both jobs have absolute deadlines at time 30, $J_{1,2}$ continues to execute until it completes at time 2, whereupon $J_{2,3}$ executes uninterrupted until time 26.

Figure 2.2: An illustration of preemptive uniprocessor scheduling. We consider a set Γ of two tasks. Task τ_1 has a WCET $C_1 = 7$ and a period $T_1 = 15$. Task τ_2 has a WCET $C_2 = 4$ and a period $T_2 = 10$. These are implicit-deadline tasks; their relative deadlines are equal to their periods.

2.2.2 Elastic Scheduling

The elastic model for implicit-deadline tasks on a uniprocessor [39, 40] characterizes each task τ_i=$(C_i, U_i^{\min}, U_i^{\max}, U_i, E_i)$ by five non-negative parameters:

- C_i: The task's worst-case execution time.
- $U_i^{\max}$: The task's maximum utilization, i.e., its nominal value when executing at the desired service level in an uncompressed state.
- $U_i^{\min}$: Its minimum utilization, i.e., a bound on the amount its service can degrade.
- U_i: The task's assigned utilization, constrained to $U_i^{\min} \leq U_i \leq U_i^{\max}$ (the value of this parameter needs to be assigned prior to run-time).
- E_i: An elastic constant, representing "the flexibility of the task to vary its utilization" [39].

A task system $\Gamma = \{\tau_1, \ldots, \tau_n\}$ has a total uncompressed utilization $U_{\text{SUM}}^{\max}$ expressed as

$$U_{\text{SUM}}^{\max} = \sum_{i=1}^{n} U_i^{\max} \tag{2.1}$$

and a desired utilization U_D representing the utilization bound allowed by the scheduling algorithm in use. In the event of system overload, i.e., if $U_{\text{SUM}}^{\max} > U_D$, the elastic model assigns a utilization U_i to each task τ_i according to these three conditions:

1. $\sum_i U_i = U_D$, i.e., total utilization is set to the schedulable bound.
2. Any task for which $E_i = 0$ is considered inelastic; this is equivalent to the case that $U_i^{\min} = U_i^{\max}$.
3. For all other tasks τ_i and τ_j, if $U_i > U_i^{\min}$ and $U_j > U_j^{\min}$, then U_i and U_j must satisfy the relationship[3]

$$\left(\frac{U_i^{\max} - U_i}{E_i}\right) = \left(\frac{U_j^{\max} - U_j}{E_j}\right) \tag{2.2}$$

[3] For tasks τ_i having $E_i = 0$, $U_i = U_i^{\min}$, and therefore the relationship needs not be satisfied.

A task system Γ for which such U_i exist for all tasks is said to be *feasible*.

Intuitively, this model represents each task as a spring, with a length corresponding to the utilization U_i and an elasticity corresponding to E_i. The total length of the springs, placed end-to-end, represents U_{SUM}. If this exceeds U_D, the schedulable bound, a force is applied to the system that compresses each task's utilization U_i proportionally to its elasticity, subject to the constraint that the utilization remains no less than the specified minimum $U_i^{\min}$.[4] This physical analogy is illustrated in Figure 2.3.

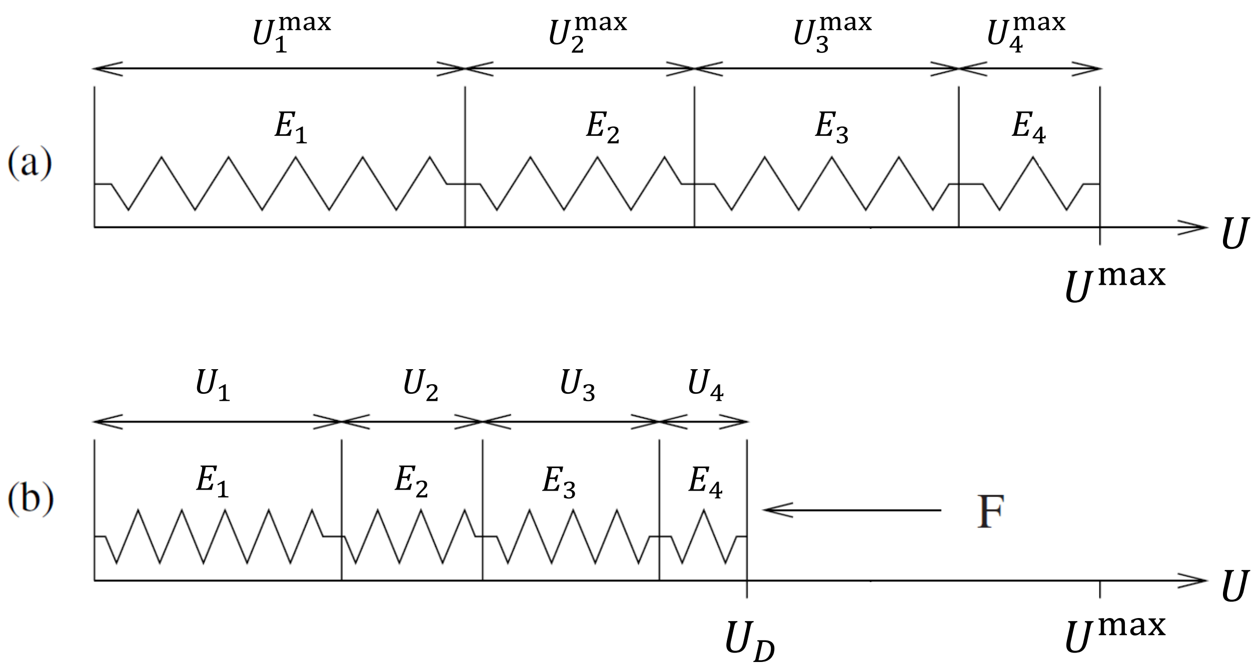


Figure 2.3: The physical spring analogy of Buttazzo's elastic model, reproduced from [37, Figure 9.27] with modifications to the labels. **(a)** shows the uncompressed task set, while **(b)** illustrates the application of elastic compression to achieve schedulability.

Compression is then realized by adjusting each task's period T_i according to its new utilization, i.e., $T_i = C_i/U_i$.

[4]This statement holds true for inelastic tasks, as $E_i = 0$ implies $U_i^{\min} = U_i^{\max}$, and therefore the utilization is not reduced.

2.2.3 Overview of the Prior Algorithm

Let Γ denote a feasible task system with $E_i > 0$ for all tasks[5] $\tau_i \in \Gamma$, and consider the U_i values that bear witness to this feasibility (i.e., each U_i either equals $U_i^{\min}$, or satisfies Equation 2.2). The tasks in Γ may be partitioned into two classes — Γ_{VARIABLE} (those tasks for which $U_i > U_i^{\min}$, and which can therefore have their utilizations compressed further if necessary) and Γ_{FIXED} (those for which $U_i = U_i^{\min}$; i.e., their utilizations are now fixed).

It has been shown [39, Equation 8] that for each $\tau_i \in \Gamma_{\text{VARIABLE}}$, the utilization U_i takes the value

$$U_i = U_i^{\max} - \left(\frac{U_{\text{SUM}} - (U_D - \Delta)}{E_{\text{SUM}}}\right) \times E_i \tag{2.3}$$

where

$$U_{\text{SUM}} = \sum_{\tau_i \in \Gamma_{\text{VARIABLE}}} U_i^{\max} \tag{2.4}$$

and

$$E_{\text{SUM}} = \sum_{\tau_i \in \Gamma_{\text{VARIABLE}}} E_i \tag{2.5}$$

respectively denote the sum of the $U_i^{\max}$ parameters and the E_i parameters of all the tasks in Γ_{VARIABLE}, and

$$\Delta = \sum_{\tau_i \in \Gamma_{\text{FIXED}}} U_i^{\min} \tag{2.6}$$

denotes the sum of the $U_i^{\min}$ parameters of all the tasks in Γ_{FIXED}.[6] Given a set of elastic tasks Γ, the algorithm of [39, Figure 3] starts out computing U_i values for the tasks assuming that they are all in Γ_{VARIABLE} — i.e., their U_i values are computed according to Equation 2.3. If any U_i so computed is observed to be smaller than the corresponding $U_i^{\min}$ then ① that task is moved from Γ_{VARIABLE} to Γ_{FIXED}; ② the values of U_{SUM}, E_{SUM}, and Δ are updated to reflect this transfer; and ③ U_i values are recomputed for all the tasks. The process terminates if no computed U_i value is observed to be smaller than the corresponding $U_i^{\min}$. It is easily seen that one such iteration (i.e., computing U_i values for all the tasks) takes $\mathcal{O}(n)$ time. Since

[5] Tasks τ_i with $E_i = 0$ must have $U_i \leftarrow U_i^{\max}$; we assume this is done in a pre-processing step, with the value of U_D updated to reflect the remaining available utilization.

[6] Observe that Δ equals the amount of utilization that is allocated to the tasks in Γ_{FIXED}; therefore $(U_D - \Delta)$ represents the amount available for the tasks in Γ_{VARIABLE}, and $\big(U_{\text{SUM}} - (U_D - \Delta)\big)$ the amount by which the cumulative utilizations of these tasks must be reduced from their desired maximums. As shown in the RHS of Equation 2.3, under elastic scheduling this reduction is shared amongst the tasks in proportion to their elasticity parameters: τ_i's share is (E_i / E_{SUM}).

an iteration is followed by another only if some task is moved from Γ_{VARIABLE} to Γ_{FIXED} and there are n tasks, the number of iterations is bounded from above by n. The overall running time for the algorithm of [39, Figure 3] is therefore $\mathcal{O}(n^2)$.

In his hard real-time computing systems textbook, Buttazzo presents a corrected and improved version of the algorithm that avoids explicitly maintaining separate lists to track Γ_{VARIABLE} and Γ_{FIXED} [37, Figure 9.29]. Nonetheless, the overall running time for the algorithm remains quadratic in the number of tasks. For reference, we present a modified version of this procedure in Algorithm 1. Notation has been modified to match our own.

Algorithm 1: Elastic_Compression(Γ, U_D) (adopted from [37, Figure 9.29])

$U_{\text{SUM}}^{\min} \leftarrow \sum_{\tau_i} C_i/T_i^{\max}$
if $U_D < U_{\text{SUM}}^{\min}$ **then return** INFEASIBLE
forall $\tau_i \in \Gamma$ **do** $U_i \leftarrow C_i/T_i^{\max}$
do
    $U_{\text{FIXED}} \leftarrow 0$, $U_{\text{VARIABLE}} \leftarrow 0$, $E_{\text{SUM}} \leftarrow 0$
    forall $\tau_i \in \Gamma$ **do**
        if $(E_i = 0)$ **or** $(T_i = T_i^{\max})$ **then**
            $U_{\text{FIXED}} \leftarrow U_{\text{FIXED}} + U_i$
        else
            $E_{\text{SUM}} \leftarrow E_{\text{SUM}} + E_i$
            $U_{\text{VARIABLE}} \leftarrow U_{\text{VARIABLE}} + U_i$
        end
    end
    OK $\leftarrow$ TRUE
    forall $\tau_i \in \Gamma$ **do**
        if $E_i > 0$ **or** $T_i < T_i^{\max}$ **then**
            $U_i \leftarrow U_i^{\max} - (U_{\text{variable}} - U_D + U_{\text{FIXED}})E_i/E_{\text{SUM}}$
            $T_i \leftarrow C_i/U_i$
            if $T_i > T_i^{\max}$ **then**
                $T_i \leftarrow T_i^{\max}$
                OK $\leftarrow$ FALSE
            end
        end
    end
while OK = FALSE
return FEASIBLE

The same algorithm was also repurposed in [39] for admission control — i.e., for determining whether a new task seeking to join an already-executing system could be admitted without compromising feasibility, and if so, recomputing the utilization values for the new task as well as for all preëxisting ones.

Extensions to elastic scheduling that were proposed by Chantem et al. [44, 45] reformulate the problem of determining the utilizations as a quadratic programming problem. This allows the iterative technique in [39] to be applied to a more general class of problems. However, this reformulation continues to have quadratic time-complexity.

In [39], Buttazzo et al. note that the quadratic time complexity is due to the enforcement of constraints on minimum utilization. If tasks are not thus constrained, the algorithm can run in linear time. Intuitively, we may consider that some tasks, representing non-critical best-effort computation, need not be characterized with minimum utilizations. However, without these constraints, the algorithm in [39, Figure 3] can assign negative utilizations, which we illustrate in the following example.

Example 1. *Consider a set* Γ *of implicit-deadline elastic tasks parameterized as in Table 2.1 that needs to be scheduled by EDF on a single processor.*

Task $\boldsymbol{\tau_i}$	$\boldsymbol{U_i^{\max}}$	$\boldsymbol{E_i}$
τ_1	0.9	1
τ_2	0.9	1
τ_3	0.2	8

Table 2.1: Elastic Tasks Compressed to Negative Utilizations

The total uncompressed utilization $U_{\text{SUM}}^{\max}$ *is* 2.0 *per Equation 2.1, but the desired utilization is* $U_D = 1.0$ *due to the utilization bound of preemptive EDF scheduling. Then, in the absence of a constraint* $U_i^{\min}$ *on the minimum utilization of each task, the utilization* U_i *of each task* τ_i *will be assigned according to Equation 2.3:*

$$U_i = U_i^{\max} - \left(\frac{2.0 - 1.0}{E_{\text{SUM}}}\right) \times E_i = U_i^{\max} - \left(\frac{1.0}{10}\right) \times E_i$$

where E_{SUM} *is calculated per Equation 2.5. Computing* U_i *for each task* τ_i*, we obtain the following utilization assignments:*

- $U_1 = 0.9 - 0.1 \times 1 = \quad 0.8$

- $U_2 = 0.9 - 0.1 \times 1 = \quad 0.8$
- $U_3 = 0.2 - 0.1 \times 8 = -0.6$

While this set of assignments does achieve a total utilization $\sum_i U_i = 1.0$ *equal to the desired value* U_D*, these assignments are not valid: a negative utilization does not have semantic meaning.*

Thus, the elastic problem *with* minimum utilization constraints $U_i^{\min}$ is the only meaningful expression of the problem in the context of task scheduling, even if the constraints are set to 0 just for the purpose of enforcing non-negative utilization assignments. Therefore, the algorithm in [39, Figure 3] cannot be guaranteed to have better than quadratic time complexity in the number of tasks.

2.3 An Improved Algorithm

We now present our improved algorithm, which runs in time quasilinear on the number of tasks. Let us first define an attribute ϕ_i for elastic task τ_i as follows:

$$\phi_i \stackrel{\text{def}}{=} \left(\frac{U_i^{\max} - U_i^{\min}}{E_i}\right) \tag{2.7}$$

We will prove a result (Theorem 1 below) that allows us to conclude that in the algorithm of [39, Figure 3], tasks may be "moved" from Γ_{VARIABLE} to Γ_{FIXED} in order of their ϕ_i parameters.

Assuming that the tasks are indexed such that $\phi_i \leq \phi_{i+1}$ for all $i, 1 \leq i < n$, we can then simply make a *single* pass through all the tasks from τ_1 to τ_n, identifying, and computing U_i values for, all the ones in Γ_{FIXED} before any of the ones in Γ_{VARIABLE}. With appropriate bookkeeping (see the pseudo-code in Algorithm 2) this can all be done in a single pass in $\mathcal{O}(n)$ time. The cost of sorting the tasks in order to arrange them according to non-increasing ϕ_i parameters is $\mathcal{O}(n \log n)$, and hence dominates the overall run-time complexity. Determining feasibility and computing the U_i parameters can therefore be done in $\mathcal{O}(n \log n) + \mathcal{O}(n) = \boldsymbol{\mathcal{O}(n \log n)}$ time.

Admission control — determining whether it is safe to add a new task and recomputing all the U_i parameters if so — requires that the new task be inserted at the appropriate location in the already sorted list of preëxisting tasks — this can be achieved in $\mathcal{O}(\log n)$ time by implementing the list as a sorted iteratable data structure such as a balanced binary tree. Once this is done, the U_i values can be recomputed in $\boldsymbol{\mathcal{O}(n)}$ time by the pseudocode in Algorithm 2. Similarly, removing a task from the system and recomputing the U_i values also takes $\mathcal{O}(n)$ time. Furthermore, if U_D changes — e.g., in response to changes in available utilization due to dynamic resource reallocation — the sorted list of tasks and their parameters do not change, and so the U_i values can be updated in linear time.

Algorithm 2: Elastic_Implicit_Uniprocessor(Γ, U_D)

Input: A list Γ of elastic tasks sorted in non-decreasing order of their ϕ_i parameters (see Equation 2.7) and a desired utilization U_D

Output: Feasibility and the list Γ with computed U_i values

$U_{\text{SUM}} \leftarrow 0$; $E_{\text{SUM}} \leftarrow 0$; $\Delta \leftarrow 0$
forall $\tau_i \in \Gamma$ **do**
  $U_{\text{SUM}} = U_{\text{SUM}} + U_i^{\max}$
  $E_{\text{SUM}} = E_{\text{SUM}} + E_i$
end
forall $\tau_i \in \Gamma$ **do**
  if $\left(U_i^{\max} - \frac{U_{\text{SUM}} - (U_D - \Delta)}{E_{\text{SUM}}} \times E_i \leq U_i^{\min}\right)$ **then**
    ▷ `Task` τ_i `is no longer compressible — it's in` Γ_{FIXED}
    $U_i \leftarrow U_i^{\min}$ ▷ `Since` $\tau_i \in \Gamma_{\text{FIXED}}$
    $\Delta \leftarrow \Delta + U_i^{\min}$ ▷ `This additional amount of utilization is allocated to tasks in` Γ_{FIXED}
    if $(\Delta > U_D)$ **then return** INFEASIBLE ▷ `Cannot accommodate the minimum requirements`
    $U_{\text{SUM}} \leftarrow U_{\text{SUM}} - U_i^{\max}$ ▷ `Since` τ_i `is removed from` Γ_{VARIABLE}
    $E_{\text{SUM}} = E_{\text{SUM}} - E_i$ ▷ `As above — since` τ_i `is removed from` Γ_{VARIABLE}
  else
    ▷ `Remaining tasks are all compressible (i.e., in` Γ_{VARIABLE}`)`
    $U_i \leftarrow U_i^{\max} - \frac{U_{\text{SUM}} - (U_D - \Delta)}{E_{\text{SUM}}} \times E_i$ ▷ `As per Equation 2.3`
  end
end
return FEASIBLE

Proof of Correctness

Theorem 1. *If* $\tau_i \in \Gamma_{\text{FIXED}}$ *and* $\phi_i \geq \phi_j$ *then* $\tau_j \in \Gamma_{\text{FIXED}}$.

Proof. Consider some iteration of the algorithm of [39, Figure 3] such that τ_i and τ_j both start out in Γ_{VARIABLE}, but τ_i is determined to belong in Γ_{FIXED} in this iteration. This implies that $U_i^{\min}$ is at least as large as the value of U_i that is computed according to Equation 2.3:

$$U_i^{\min} \geq U_i^{\max} - \left(\frac{U_{\text{SUM}} - (U_D - \Delta)}{E_{\text{SUM}}} \right) \times E_i$$

By algebraic simplification of the above, we have

$$\left(\frac{U_{\text{SUM}} - (U_D - \Delta)}{E_{\text{SUM}}} \right) \geq \left(\frac{U_i^{\max} - U_i^{\min}}{E_i} \right) \tag{2.8}$$

Note that the LHS of Expression 2.8 does not contain any term specific to τ_i and so is the same for all the tasks in Γ_{VARIABLE} for this iteration, and that the RHS is simply ϕ_i. Since $\phi_i \geq \phi_j$ (as per the statement of the theorem), we may conclude by the transitivity of the $\geq$ operator on the real numbers that the LHS of Expression 2.8 would also be $\geq \phi_j$; equivalently, the value of $U_j^{\min}$ is no smaller than the value of U_j that is computed according to Equation 2.3, and as a consequence τ_j, too, should be moved to Γ_{FIXED} . □

2.4 Evaluation

In this section, we compare the performance of our improved algorithm for elastic scheduling of implicit-deadline tasks on a uniprocessor outlined in Algorithm 2 to the algorithm from Buttazzo et al. [37, Figure 9.29] listed in Algorithm 1.

2.4.1 Implementation

Evaluations are performed on a Raspberry Pi 3 Model B+, which has a Broadcom BCM2837B0 System on Chip (SoC) with a 4-core ARMv8 Cortex-A53 running at 700 MHz[7] and 1GB of RAM. We used version 6.1.21 of the Linux kernel, compiled for the ARMv7l 32-bit ISA. We implement both algorithms in C++ and quantify execution time performance by measuring elapsed processor cycles. We read directly from the cycle counter using a custom driver and kernel module that enables access to the ARM performance monitoring unit (PMU) from userspace. Algorithms are compiled statically using version 10.2.1 of the Gnu Compiler Collection (GCC) at optimization level `O0`; this allows us to avoid undesirable instruction reordering, especially around reads to the cycle counter. To avoid interference from other processes, we disable real-time throttling by writing -1 to the file `/proc/sys/kernel/sched_rt_runtime_us`, isolate CPU core 3 from the scheduler, and run our algorithms on that core at the highest real-time priority under Linux's `SCHED_FIFO` scheduling class.

Each task τ_i is represented as a data structure (`struct`) containing single-precision floating-point representations of $U_i^{\max}$, $U_i^{\min}$, U_i, and E_i. The derived parameter ϕ_i from Equation 2.7 is also an attribute of the structure. For these experiments, we are only concerned with the assignment of U_i values; we therefore do not represent the WCET C_i or period T_i of any task.

For Buttazzo's algorithm (Algorithm 1), since we are considering only utilization assignments, and not period updates, we do not implement Line 18. Line 3 is also not implemented, as we assign $U_i \leftarrow U_i^{\max}$ when task parameters are generated. Line 20 is implemented as $U_i \leftarrow U_i^{\min}$ instead. Furthermore, all period-related checks (Lines 7, 16, and 19) are converted to the equivalent comparison of U_i to $U_i^{\min}$ or $U_i^{\max}$.

For our improved algorithm (Algorithm 2), we compare three implementations:

1. The set of tasks Γ is implemented as an array (`std::vector`), which is sorted prior to executing the algorithm. Inserting or removing tasks takes linear time to move array

[7] The processor supports a CPU clock speed of 1.4 GHz. However, as noted in [26], this frequency cannot be sustained continuously, and may lead to throttling and instability. To maintain predictability, we boot the Raspberry Pi with a constant 700 MHz CPU clock speed, set the GPU to 250 MHz, and disable throttling. Details can be found at https://www.raspberrypi.com/documentation/computers/config_txt.html

elements (and, in the case of insertion, to find the location to insert to maintain sorted order).

2. The set of tasks Γ is implemented as a balanced binary tree (`std::set`), sorted by ϕ_i. Constructing the set takes quasilinear time, but subsequent insertion and removal requires only logarithmic time, while enabling sequential iteration over tasks in sorted order.

3. The set of tasks Γ is implemented as a linked list (`std::list`), sorted by ϕ_i. Removing a task takes constant time, but adding a task takes linear time to find the location to insert in sorted order.

2.4.2 Generating Task Sets

We generate sets Γ of tasks τ_i using the following methodology:

1. We consider sets of n tasks, generating 10 000 sets for each value of n in 2–50.

2. Each set of tasks has a total maximum utilization $U_{\text{SUM}}^{\max}$ selected at random uniformly from $(1.0, 2.0]$ and a total minimum utilization $U_{\text{SUM}}^{\min}$ selected at random uniformly from $(0.0, 1.0]$.

3. We apply the Dirichlet Rescale (DRS) algorithm [70] to distribute the total maximum utilization $U_{\text{SUM}}^{\max}$ in an unbiased random fashion across the $U_i^{\max}$ values for each individual task. We note that, in this context, the result should be equivalent to an application of the earlier UUniFast and UUniSort techniques [24].

4. We then apply the DRS algorithm to distribute the total minimum utilization $U_{\text{SUM}}^{\min}$ across the individual $U_i^{\min}$ values. DRS allows us to select these values uniformly from the space of selections satisfying the conditions that *(i)* the total $\sum_i U_i^{\min}$ equals the specified $U_{\text{SUM}}^{\min}$ and *(ii)* each value $U_i^{\min}$ does not exceed the corresponding $U_i^{\max}$.

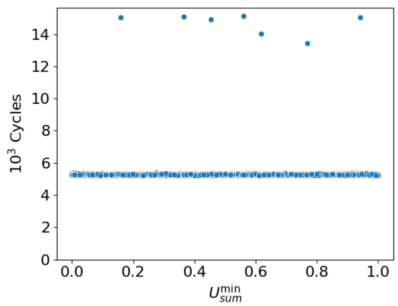
5. Each task τ_i is assigned an elasticity E_i at random, selected uniformly from the range $(0, 1]$.

2.4.3 Execution Time of Compression for Schedulability

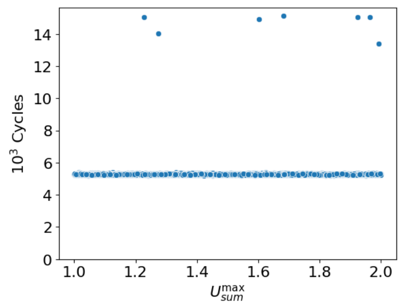
We execute our implementations of Algorithms 1 and 2 for each set of tasks thus generated to compress the set of tasks to a total utilization of 1.0 (to be EDF-schedulable on a single processor). We measure execution time by reading directly from the cycle counter, reporting elapsed CPU cycles. We separately measure the initialization and compression times for each algorithm. For Buttazzo's procedure in Algorithm 1 , initialization involves computing the $U_{\text{SUM}}^{\min}$ value and checking whether it exceeds U_D, while compression is the **do . . . while** loop in the algorithm. For our improved algorithm in Algorithm 2, compression is the **forall** loop that iterates over tasks in order of their ϕ_i parameters. The dominant contribution to the algorithm's execution time complexity is the sorting of tasks by their ϕ_i values; we therefore include in initialization time both the computation of U_{SUM} and E_{SUM} as well as the total time to calculate each task's ϕ_i value and establish the sorted order. For the array and list, the sort is performed over the complete data structure; for the binary tree, we insert tasks individually as their ϕ_i values are calculated. The results allow us to answer the following questions.

Do the values selected for $U_{\text{SUM}}^{\max}$ or $U_{\text{SUM}}^{\min}$ affect execution time?

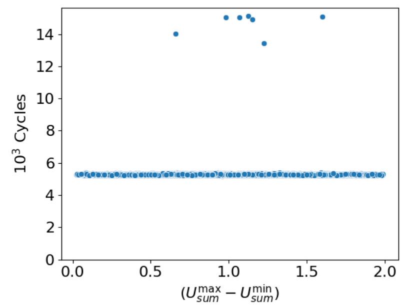
For the 10 000 sets of 50 tasks generated, we produce 3 scatter plots for the initialization ("Init") and compression ("Compress") times of each implementation: these plot cycles against the values $U_{\text{SUM}}^{\min}$, $U_{\text{SUM}}^{\max}$, and the absolute distance between them, $\left(U_{\text{SUM}}^{\max} - U_{\text{SUM}}^{\min}\right)$. These are shown in Figure 2.4. ***We do not observe a significant dependence of execution time on the minimum and maximum utilization of the task set***. As such, in our subsequent evaluations, we consider data in aggregate, rather than grouping by utilization values.



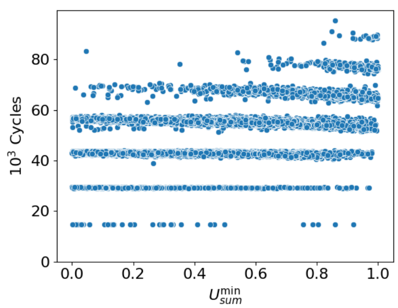
(a) Buttazzo Init.



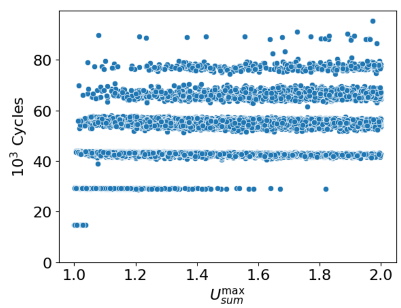
(b) Buttazzo Init.



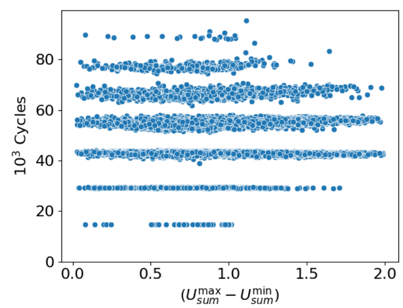
(c) Buttazzo Init.



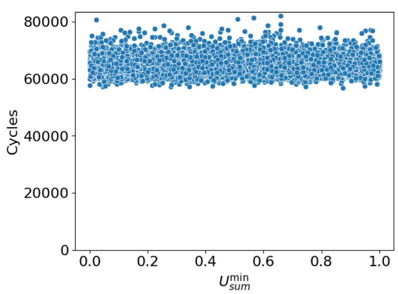
(d) Buttazzo Compress.



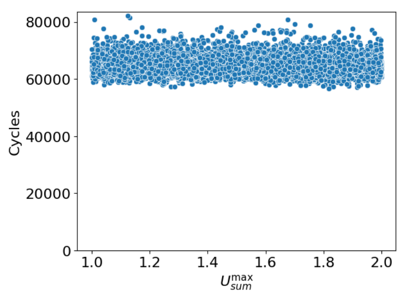
(e) Buttazzo Compress.



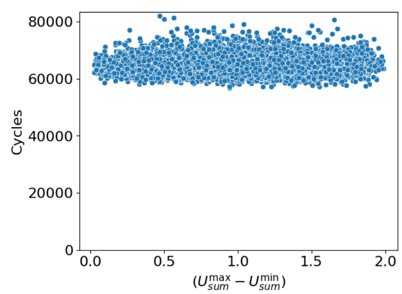
(f) Buttazzo Compress.



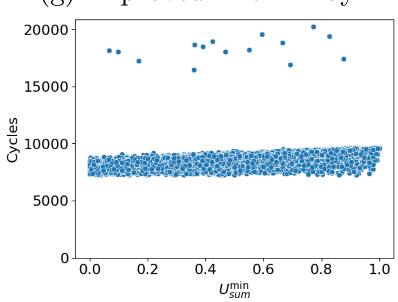
(g) Improved Init: Array.



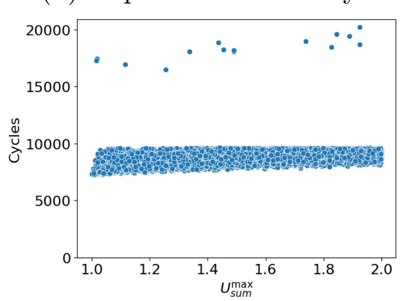
(h) Improved Init: Array.



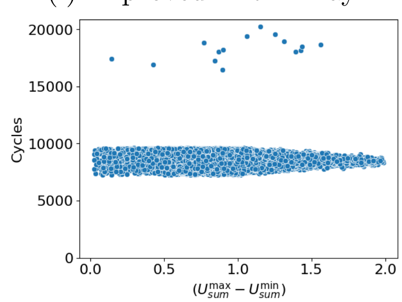
(i) Improved Init: Array.



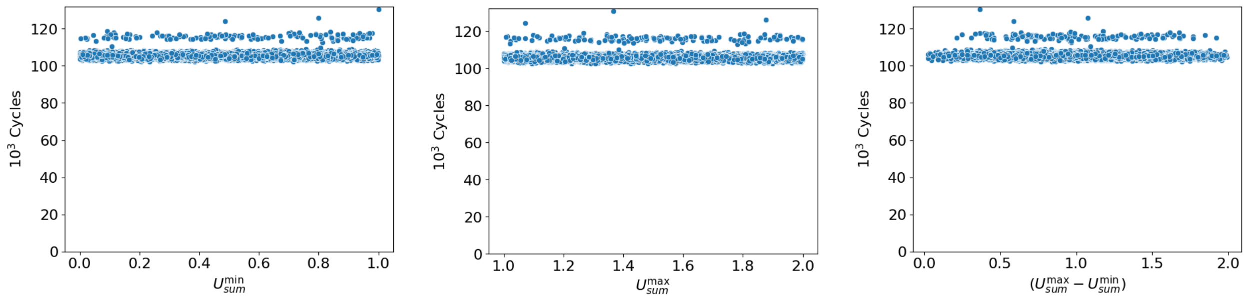
(j) Improved Compress: Array.



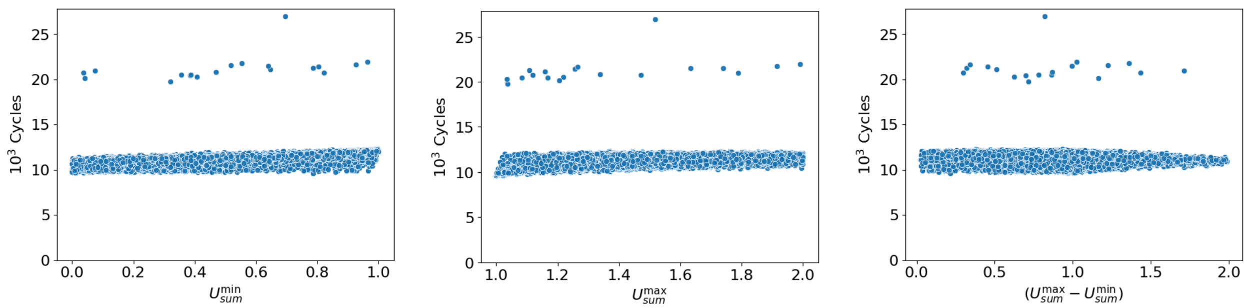
(k) Improved Compress: Array.



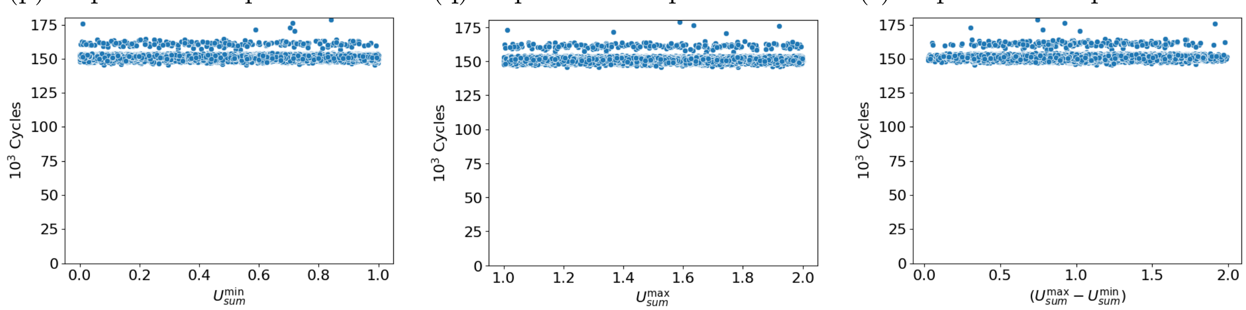
(l) Improved Compress: Array.



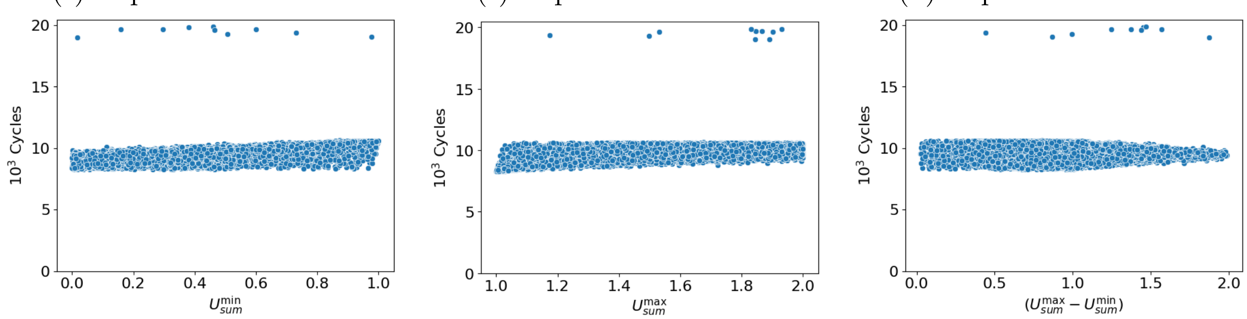
(m) Improved Init: Tree. (n) Improved Init: Tree. (o) Improved Init: Tree.



(p) Improved Compress: Tree. (q) Improved Compress: Tree. (r) Improved Compress: Tree.



(s) Improved Init: List. (t) Improved Init: List. (u) Improved Init: List.



(v) Improved Compress: List. (w) Improved Compress: List. (x) Improved Compress: List.

Figure 2.4: Execution times by utilization metrics for 50 tasks.

How well does our improved algorithm perform in reality?

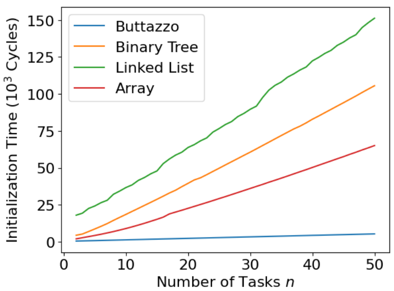
We separately measure the mean, median, and maximum execution times to initialize ("Init") and compress ("Compress") the 10 000 sets of tasks generated for each size n from 2–50. These times, as well as the total, are reported in Figure 2.5.

As expected, the time to initialize Buttazzo's algorithm (Algorithm 1) is much faster than our improved algorithm (Algorithm 2), which has to sort tasks by their ϕ_i values. Of our three implementations of our algorithm, the linked list was the slowest to initialize, while the array was the fastest; we assume that this was due to the data locality and simplicity of managing the data structure.

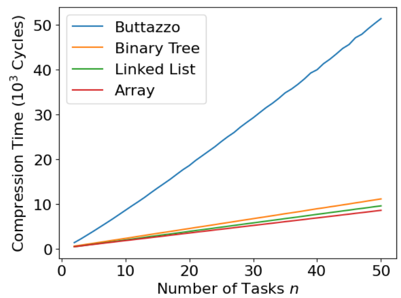
Also, as expected, ***the compression time for Buttazzo's algorithm was much longer than for our algorithm***. On average, the array tended to be the fastest, followed by the linked list, followed by the binary tree.[8] This makes sense; while all three data structures enable linear time traversal, the array is the simplest to iterate (in fixed-size strides) and has the best data locality; the linked list is still simple, but requires following pointers between nodes, and does not have as good of locality; and the binary tree requires even more complex pointer chasing.

Most interesting, we observe that our algorithm does not strictly dominate Buttazzo's algorithm in total running time. In fact, in the average case, Buttazzo's algorithm performs better because of the low initialization overhead. In the worst case, both Buttazzo's algorithm and the array-based implementation of our algorithm dominate the other two implementations, but neither clearly dominates the other. This is partially explained by the fact that the compression times for Buttazzo's algorithm grow roughly linearly with the number of tasks, rather than quadratically, for sets of up to 50 tasks. Under non-pathological cases, the outer loop (Line 6 of Algorithm 2) of Buttazzo's algorithm only runs a handful of times.

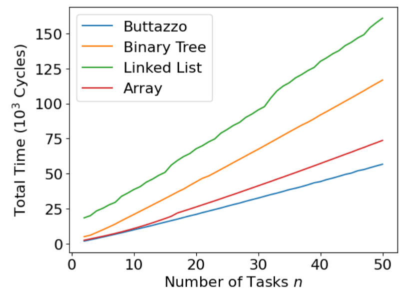
[8] We do not observe a significant difference in the trends of worst-case compression times versus number of tasks between each of the three implementations.



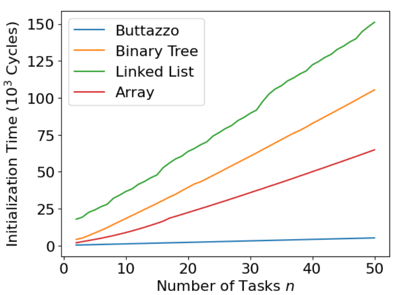
(a) Mean Init Times.



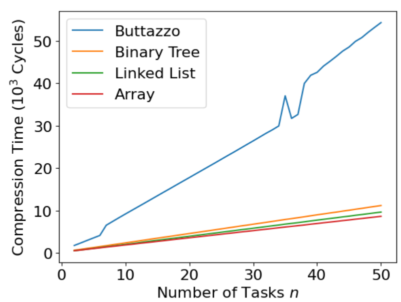
(b) Mean Compress Times.



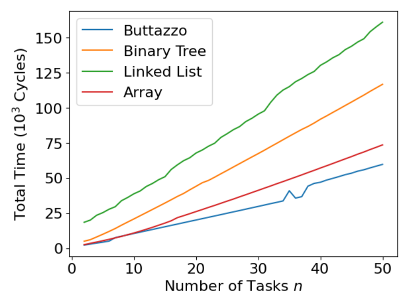
(c) Mean Total Times.



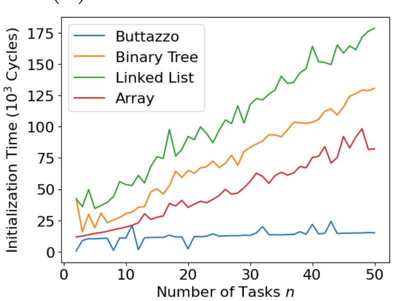
(d) Median Init Times.



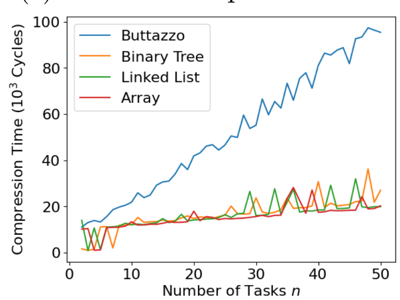
(e) Median Compress Times.



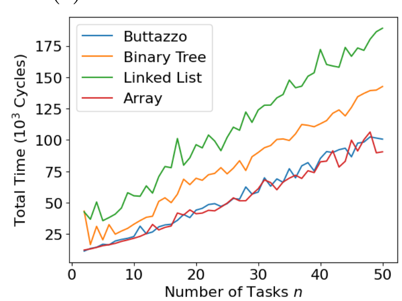
(f) Median Total Times.



(g) Max Init Times.



(h) Max Compress Times.



(i) Max Total Times.

Figure 2.5: Performance scaling with number of tasks.

Nonetheless, we argue that our algorithm is better in practice. While there is not a clear advantage to using our algorithm to perform compression over a complete set of tasks, there is no clear *disadvantage* either. Furthermore, ***our algorithm performs better in situations where initialization has already happened***, e.g. for online adjustment in response to changes in available utilization. In this case, when only compression needs to occur, associated overheads are summarized in Table 2.2. The worst execution times that we observed for the array-based implementation of our algorithm were 3.45× faster than those of Buttazzo's algorithm when just compressing tasks.

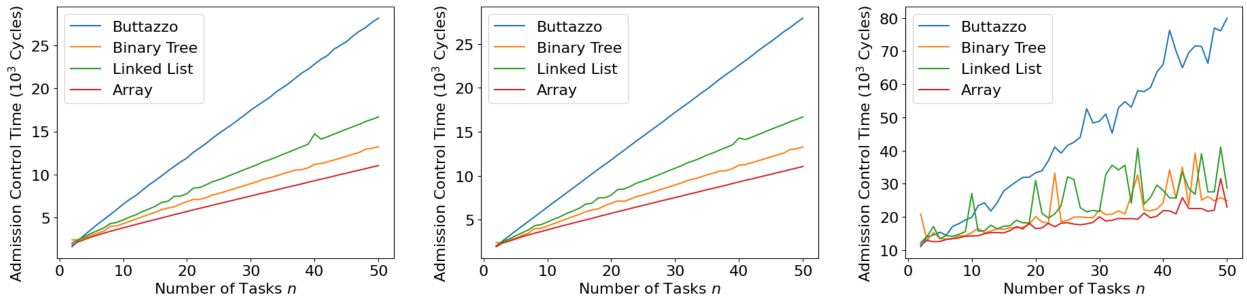
	Buttazzo	Binary Tree	Linked List	Array
mean	51 380	11 157 (4.61×)	9 617 (5.34×)	8 616 (5.96×)
median	54 315	11 175 (4.86×)	9 629 (5.64×)	8 626 (6.30×)
maximum	97 342	36 231 (2.69×)	31 984 (3.04×)	28 202 (3.45×)

Table 2.2: Greatest mean, median, and maximum **compression times** (cycles) observed for up to 50 tasks. Values in parentheses indicate speedup compared to Buttazzo's algorithm.

Furthermore, as we will demonstrate, there is a clear advantage to using our algorithm during online admission of a new task.

2.4.4 Execution Time of Task Admission

We modify our implementations of Algorithms 1 and 2 to measure admission of a single task. For the sets of n tasks of size 2–50 that we already generated, we apply each algorithm to the first $n-1$ tasks. We then measure the time to compress them after adding the final task from the set. For Buttazzo's algorithm, this requires rerunning the complete **do . . . while** loop. For our algorithm, this requires ① computing the value ϕ_i of the new task τ_i according to Equation 2.7, then ② adding its utilization and elasticity to U_{SUM} (Equation 2.4) and E_{SUM} (Equation 2.5) or Δ (Equation 2.6), as appropriate. The task is then ③ inserted into the sorted container, before finally ④ executing the **forall** loop in Algorithm 2.



(a) Mean Admission Times. (b) Median Admission Times. (c) Max Admission Times.

Figure 2.6: Execution time for admitting the n^{th} task.

Results are illustrated in Figure 2.6. We observe that, when admitting a new task, all implementations of our improved algorithm dominate Buttazzo's algorithm for more than 3 tasks in the average case, and more than 10 tasks in the worst case. The array (which enables logarithmic time search for the location to insert the new task, then requires linear time to perform the insertion) performs the best on average, followed by the balanced binary tree (which allows logarithmic-time insertion, but requires pointer chasing), then the linked list (which allows constant-time insertion after linear time search for the insert location). Overheads and speedups are summarized in Table 2.3. The array-based implementation of our algorithm admits tasks 2.53× faster than Buttazzo's algorithm in the worst case.

	Buttazzo	Binary Tree	Linked List	Array
mean	28 165	13 246 (2.13×)	16 704 (1.69×)	11 043 (2.55×)
median	27 931	13 274 (2.10×)	16 698 (1.67×)	11 053 (2.53×)
maximum	79 967	39 213 (2.04×)	41 044 (1.95×)	31 566 (2.53×)

Table 2.3: Greatest mean, median, and maximum task **admission times** (cycles) observed for up to 50 tasks. Values in parentheses indicate speedup compared to Buttazzo's algorithm.

2.5 Conclusions

In this chapter, we have presented a new approach to elastic scheduling of implicit-deadline tasks. Our proposed algorithm, listed in Algorithm 2, achieves compression in time quasi-linear in the number of tasks. Furthermore, it runs in linear time when adjusting online to dynamic changes in system state, e.g., during admission of a new task, or when available utilization decreases. Compared to Buttazzo's algorithm in [39, 40], which runs in time quadratic in the number of tasks, ***our approach has better time complexity.***

In practice, we have demonstrated that Buttazzo's original algorithm [37, Figure 9.29] for elastic scheduling [39, 40] is significantly slower to compress tasks compared to our proposed approach in Algorithm 2. However, initializing a sorted data structure dominates the execution time of Algorithm 2. As a result, for smaller numbers of tasks (up to 50), there is no clear advantage to using our algorithm over that of Buttazzo (or vice versa) to compress complete task sets for schedulability.

However, we observed that ***our algorithm achieves significant speedup during admission of new tasks.*** Because admission control involves online scheduling decisions, it is especially important that its overhead remains bounded. Therefore, in situations where initialization has already occurred — e.g., during admission of a new task or in response to changes in available utilization — Algorithm 2 is significantly faster than Buttazzo's approach. As these are more likely to be the scenarios encountered during dynamic online scenarios where the adaptation must have predictably low overheads, our proposed approach has clear advantages.

Chapter 3

Efficient Algorithms for Multiprocessors

3.1 Introduction

Buttazzo's elastic scheduling model [39, 40] only considered *uniprocessor* scheduling of implicit-deadline tasks. The growing prevalence of multicore CPUs, even in embedded platforms, have enabled the deployment of applications that utilize the increase in available processor resources. This has prompted a growing body of work in the analysis of algorithms that schedule sets of sequential tasks on *multiple processors*.

Though recent advances in multicore embedded platforms have made lightweight, low-cost, yet very capable systems highly accessible, constraints on size, weight, and power (SWaP) nonetheless often limit the number of tasks that can be safely scheduled on such systems. Orr and Baruah therefore extended elastic scheduling theory to multiprocessors [118], enabling task systems to adapt to overload on multicore platforms. They considered both the global and partitioned scheduling paradigms, proposing algorithms for elastic scheduling of tasks under the fluid [19], global EDF [93], PriD [69], and partitioned EDF schedulers [99]. Their model retains Buttazzo's semantics of proportional compression, with constraints on each task's minimum utilization. However, instead of compressing to a utilization bound, they extend the model to apply compression *until the system is schedulable*; this provides more generality for multiprocessor scheduling models for which feasibility is not simply verified according to total utilization.

Orr and Baruah evaluate the ability of elastic scheduling to find feasible utilization assignments for each considered scheduling paradigm; however, they do not assess the execution time of their proposed algorithms. In this chapter, we consider two of the multiprocessor elastic algorithms they extended — *fluid* and *partitioned* EDF scheduling — with the goal of improving their execution time performance. The intention of this chapter is to use empirical evidence to provide guidance toward the selection of an appropriate elastic scheduling approach in the context of given scenarios.

Under the fluid abstraction, a set of tasks is deemed schedulable if *(i)* their total utilization does not exceed the number of processor cores, and *(ii)* no individual task's utilization exceeds 1. We therefore demonstrate that it is straightforward to extend our improved algorithm for implicit-deadline elastic scheduling — Algorithm 2 in Chapter 2 — to fluid scheduling by using the number of processor cores as the utilization bound.

Partitioned EDF scheduling is equivalent to the bin-packing problem, and is therefore NP-hard [99]. Orr and Baruah extend elastic scheduling to partitioned EDF by introducing a term representing the "amount" of compression to be applied to the task system [118]. They iteratively increase this term at a specified granularity, at each step checking feasibility using three bin-packing heuristics; if any one succeeds, the algorithm terminates. We consider three alternative approaches. First, the order in which heuristics are attempted is modified according to their ability to find a suitable partition. Second, we employ a binary, rather than iterative, search over the space of possible compression constrained by the minimum utilizations of each task. Third, using the insight that under partitioned EDF scheduling, a set of tasks is guaranteed to be schedulable if its utilization does not exceed a function of the number of cores, we apply Algorithm 2 for compressing to this utilization bound.

For each proposed approach to elastic scheduling for partitioned EDF, we compare the trade-offs imposed by faster, yet more pessimistic, compression. Through extensive evaluation, we demonstrate that using a binary search over the amount of compression, while changing the application of partitioning heuristics, improves execution time without significant overcompression. Furthermore, applying Algorithm 2 may be beneficial when rapid adaptation is necessary.

The remainder of this chapter is organized as follows:

- Section 3.2 develops the system model used in this chapter and provides necessary background on existing elastic models for fluid and partitioned EDF scheduling.

- Section 3.3 discusses the straightforward application of our improved algorithm for elastic scheduling of implicit-deadline tasks on a uniprocessor (Algorithm 2) to fluid scheduling.
- Section 3.4 proposes three alternative approaches to elastic scheduling of partitioned EDF tasks.
- Section 3.5 evaluates these in the context of Orr and Baruah's original partitioned EDF algorithm [118].
- Section 3.6 concludes the chapter.

3.2 The Multiprocessor Elastic Scheduling System Model

In [118], Orr and Baruah extended elastic models to scheduling of sequential, implicit-deadline tasks under the global and partitioned multiprocessor scheduling paradigms. In these models, tasks are parameterized as in Section 2.2.2. But unlike Buttazzo's original elastic scheduling model [39, 40], tasks are provided multiple processor cores on which to execute.

We state the problem as follows: Given a set Γ of n implicit-deadline tasks to execute preemptively on m cores, assign the maximum utilization U_i to each task τ_i that satisfies these conditions:

1. The task system is schedulable on m cores according to the selected algorithm.
2. Any task for which $E_i = 0$ is considered inelastic; we consider this equivalent to the case that $U_i^{\min} = U_i^{\max}$.
3. For all other tasks τ_i and τ_j, if $U_i > U_i^{\min}$ and $U_j > U_j^{\min}$, then U_i and U_j must satisfy the relationship of Equation 2.2.

Conditions (2) and (3) match those of Buttazzo's model (listed in Section 2.2.2). Notice, however, that condition (1) has become more general to capture the semantics of multiprocessor schedulability, while remaining applicable to a single processor if m=1.

In [118], Orr and Baruah extended elastic scheduling to the fluid, global EDF, and PriD global scheduling paradigms, as well as to partitioned EDF. Under global multiprocessor scheduling of recurrent tasks, individual tasks are not restricted to executing upon specific processors. Instead, any active job may execute upon any available processor, and a preempted job may later resume execution on a different processor. This differs from partitioned multiprocessor scheduling, under which each task is assigned to a specific core in many-to-one fashion. In other words, jobs from a single task can only execute on the single core to which they are assigned, but multiple tasks may be assigned to the same processor.

3.2.1 Fluid Scheduling

Under the fluid scheduling paradigm [19], individual tasks are assigned a fraction f of a processor at each instant in time. Implementations exist to approximate it, e.g., under the RT-FAIR scheduling framework in LITMUSˆRT [41]. It is a convenient abstraction that considers a set Γ of tasks τ_i to be schedulable on m cores so long as the following conditions hold:

1. The total utilization $\sum_i U_i$ of Γ does not exceed m.
2. The individual utilizations U_i of each task τ_i do not exceed 1.

Orr and Baruah [118] demonstrated a straightforward extension of Buttazzo's model [39, 40] to fluid scheduling. So long as the maximum utilization $U_i^{\max}$ for every task τ_i does not exceed 1, a simple application of [39, Figure 3] (reproduced as Algorithm 1) with $U_D{=}m$ will assign the correct utilization to each task.

3.2.2 Partitioned EDF

While fluid scheduling is a convenient abstraction, it often remains impractical in real systems [118]. Partitioned scheduling provides a more practical paradigm. Under partitioned EDF scheduling, each task is assigned to a single processor core, though each core may be assigned multiple tasks. On an individual core, jobs are prioritized according to their absolute deadlines — in other words, each core schedules its tasks in an EDF manner independently of the other cores. The problem of deciding whether a set of tasks are schedulable on m cores under partitioned EDF can be stated as follows:

> Given a set Γ of n tasks τ_i, each having utilization U_i, is there a partition of tasks into m sets such that the sum of utilizations in any set does not exceed 1?

This is equivalent to the bin-packing problem, and is therefore NP-hard in the strong sense. Nonetheless, there exist heuristic algorithms to solve bin-packing problems, and Lopez et al. have compared several in the context of partitioned EDF scheduling [99].

The elastic version of the problem, proposed by Orr and Baruah [118], applies this statement of partitioned EDF schedulability to condition (1) listed above for multiprocessor elastic scheduling. They observe that the degree by which compression is applied to a task system can be quantified by the relationship in Equation 2.2. In doing so, they introduce a term λ that is representative of this relationship, and express the utilization U_i of each task τ_i as:

$$U_i(\lambda) \quad \stackrel{\text{def}}{=} \quad \max\left(U_i^{\max} - \lambda E_i, U_i^{\min}\right) \tag{3.1}$$

The value of λ beyond which the utilization U_i of task τ_i takes its minimum value $U_i^{\max}$ can therefore be derived as follows:

$$U_i^{\min} = U_i^{\max} - \lambda E_i \quad \rightarrow \quad \lambda = \left(\frac{U_i^{\max} - U_i^{\min}}{E_i}\right)$$

which is equal to the value ϕ_i in Equation 2.7. As such, we may hereafter refer to ϕ_i interchangeably as $\lambda_i^{\max}$. For a complete set of tasks Γ we also define the term $\lambda^{\max}$ as the maximum compression that may be applied to the task system:

$$\lambda^{\max} \quad \stackrel{\text{def}}{=} \quad \max_{\tau_i}\left(\frac{U_i^{\max} - U_i^{\min}}{E_i}\right) = \max_{\tau_i}\left(\lambda_i^{\max}\right) \tag{3.2}$$

The problem of elastic scheduling under Buttazzo's model [39, 40] can therefore be reduced to the problem of finding the minimum value of λ for which a set of tasks are schedulable. When applied to partitioned EDF scheduling, Orr and Baruah [118] observe that finding the *optimal* value of λ to guarantee schedulability is NP-hard. Instead, they propose an approximate search technique that iterates over values of λ in the interval $[0, \lambda^{\max}]$ with some "granularity" ϵ. For each value of λ, they assess schedulability by attempting three of the heuristics for partitioned EDF from [99]; if any one deems feasibility, the algorithm terminates. They employ the "first fit," "worst fit," and "best fit" heuristics, with tasks τ_i considered in order of decreasing utilization $U_i(\lambda)$.

- **First fit:** Each task considered in order is assigned to the first processor on which it fits.
- **Worst fit:** Each task considered in order is assigned to the processor with the maximum remaining capacity.
- **Best fit:** Each task considered in order is assigned to the processor with the minimum remaining capacity on which it fits.

For n tasks on m cores, sorting tasks and partitioning them with each heuristic takes at most $\Theta(n \log n + n \cdot m)$ time. As this must be performed for each tested value of λ — of which there are up to $\left(\lfloor \frac{\lambda^{\max}}{\epsilon} \rfloor + 1\right)$ — the overall complexity is:

$$\Theta\left(\frac{\lambda^{\max}}{\epsilon} \cdot (n \log n + n \cdot m)\right)$$

3.3 Fluid Scheduling

3.3.1 Extension of the Efficient Algorithm

As discussed in Section 3.2.1, Orr and Baruah [118] demonstrated a straightforward extension of Buttazzo's model [39, 40] to fluid scheduling by proportionally compressing task utilizations from their total maximum utilization $U_{\text{SUM}}^{\max}$ to the schedulable bound $U_D = m$, where m is the number of processor cores. This is achieved by running procedure ELASTIC_COMPRESSION [39, Figure 3] (reproduced in this dissertation as Algorithm 1).

As we showed in Chapter 2, our efficient procedure ELASTIC_PARTITIONED_EDF in Algorithm 2 assigns the same utilizations as Buttazzo's algorithm. It can therefore be applied to fluid scheduling to achieve compression in $\boldsymbol{\mathcal{O}(n \log n)}$ time, or to update utilization assignments in **linear time** during admission of a new task, removal of a task, or a change in the number of available processor cores.

3.3.2 Applicability of Uniprocessor Results

Elastic compression for task sets under fluid scheduling needs only to be applied when the maximum total task utilization exceeds the number of available processor cores; i.e., when $U_{\text{SUM}}^{\max} > m$. And it can find a feasible assignment of utilizations only when the minimum total task utilization does not exceed the number of cores; i.e., when $U_{\text{SUM}}^{\min} < m$. This differs semantically from elastic scheduling of uniprocessor EDF tasks only in the value chosen for m; in the uniprocessor case, m is 1.

In Section 2.4.3, we demonstrated no significant relationship between execution time and the values $U_{\text{SUM}}^{\max}$ and $U_{\text{SUM}}^{\min}$, nor the distance between them. Therefore, similar conclusions may be drawn for fluid scheduling as for uniprocessor scheduling of implicit deadline tasks. In Sections 2.4.3 and 2.4.4, we demonstrated that our improved algorithm (Algorithm 2) is significantly faster to compress tasks compared to Buttazzo's original algorithm in [39, Figure 3] for the uniprocessor case after initializing the sorted data structure. This suggests that our improved algorithm should be applied to fluid scheduling when compressing to adapt during online execution, such as when the number of available processor cores change, or when admitting a new task.

3.4 Partitioned EDF

In this section, we propose three alternative approaches to elastic scheduling of partitioned EDF tasks. First, the order in which heuristics are applied is modified according to their ability to find a suitable partition. Second, we consider a binary, rather than iterative, search over the space of compression allowed due to the minimum utilization constraint on each task. Third, using the insight that under partitioned EDF scheduling, a set of tasks is guaranteed to be schedulable if its utilization does not exceed a function of the number of cores, we apply Algorithm 2 for compressing to this utilization bound.

3.4.1 Heuristic Selection and Order

In [118], Orr and Baruah determine whether a feasible partition exists for each tested value of λ using the first fit, worst fit, and best fit heuristics. Their reasoning is that using three heuristics instead of just one increases the chances of identifying a feasible partition, while the additional operations imply a constant multiplier to execution time, and therefore do not affect the execution time *complexity*.

Pragmatically, however, achieving even a constant-time speedup may be important for deployment to a real system. To this end, we consider whether all three heuristics are needed. Indeed, the worst fit achieves a lower theoretical bound on utilization for a given number of processor cores [12]. Eliminating worst fit achieves faster execution, and may be appropriate if any task sets it deems feasible are also feasible using the first fit or best fit heuristics. In Section 3.5.3, we empirically demonstrate this to be true for a large number of synthetic task sets.

Furthermore, we propose to change the order in which heuristics are considered. We demonstrate in Section 3.5.3 that the best fit heuristic is slightly more likely to identify a feasible partition than the first fit for the synthetic task sets considered, and applying it first may enable slightly earlier termination of the algorithm.

3.4.2 Binary Search

We observe that a straightforward optimization may be applied to the approach of Orr and Baruah [118] summarized above. Rather than iterating over all values of $\lambda \in [0, \lambda^{\max}]$ with granularity ϵ in *sequential order*, we can instead perform a *binary search* in time $\Theta\left(\log \frac{\lambda^{\max}}{\epsilon}\right)$, as outlined in Algorithm 3. Thus, total time complexity is reduced to

$$\boldsymbol{\Theta\left((n \log n + n \cdot m) \cdot \log\left(\frac{\lambda^{\max}}{\epsilon}\right)\right)} \tag{3.3}$$

Algorithm 3 uses the notation $\Gamma(\lambda)$ from Baruah [13], denoting the task system obtained from Γ by applying compression λ, i.e., with each task τ_i having a utilization $U_i(\lambda)$ according to Equation 3.1. The algorithm first checks if $\Gamma(0)$ — the uncompressed task set — is

Algorithm 3: Elastic_Partitioned_EDF(Γ, m)

Input: A list Γ of elastic tasks to schedule on m processor cores
Output: The value λ to obtain feasibility

$\lambda^{\max} \leftarrow 0$
forall $\tau_i \in \Gamma$ **do**
  $\lambda_i^{\max} \leftarrow \frac{U_i^{\max} - U_i^{\min}}{E_i}$
  $\lambda^{\max} \leftarrow \max\left(\lambda^{\max}, \lambda_i^{\max}\right)$
end
if $\Gamma(0)$ *is schedulable on m cores* **then return** *0*
if $\Gamma(\lambda^{\max})$ *is **not** schedulable on m cores* **then return** INFEASIBLE
$\lambda_{\text{HI}} \leftarrow \lambda^{\max}$
$\lambda_{\text{LO}} \leftarrow 0$
do
  $\lambda \leftarrow \left(\lambda_{\text{HI}} - \lambda_{\text{LO}}\right)/2$
  if $\Gamma(\lambda)$ *is schedulable on m cores* **then** $\lambda_{\text{HI}} \leftarrow \lambda$
  else $\lambda_{\text{LO}} \leftarrow \lambda$
while $\lambda_{\text{HI}} - \lambda_{\text{LO}} > \epsilon$
return λ_{HI}

schedulable by partitioned EDF on m cores; schedulability may be determined according to the heuristics employed by Orr and Baruah [118]. If so, it returns the value $\lambda = 0$. It then checks if $\Gamma(\lambda^{\max})$ is schedulable; if not, the algorithm fails. Otherwise, it performs binary search over values of λ in the range $[0, \lambda^{\max}]$: λ_{HI} (initialized to $\lambda_{\max}$) tracks the smallest value of λ tested for which $\Gamma(\lambda)$ is schedulable, while λ_{LO} (initialized to 0) tracks the largest tested value for which $\Gamma(\lambda)$ is *not* schedulable. At each step, the algorithm checks schedulability of $\Gamma(\lambda)$; if feasibility is determined, λ_{HI} is decreased to the tested value of λ; otherwise, λ_{LO} is increased to the tested value of λ. The algorithm terminates when the difference between λ_{HI} and λ_{LO} does not exceed ϵ.

Optimality of Search for λ

We now discuss and prove results about the optimality of iterative and binary searches for partitioned EDF scheduling. We begin by introducing the term $\lambda^*_{\Gamma,m}$, defined as the smallest value of λ for which $\Gamma(\lambda)$ is schedulable by partitioned EDF on m cores.

The first result is intuitive: it says that, once you compress a task system such that it is schedulable, it will remain schedulable when compressed more.

Theorem 2. *Given a value of λ, if $\Gamma(\lambda)$ is partitioned EDF schedulable on m cores, then $\Gamma(\lambda')$ is also partitioned EDF schedulable for every value of $\lambda' \geq \lambda$.*

Proof. Consider a set Γ of n tasks τ_i. If $\Gamma(\lambda)$ is partitioned EDF schedulable on m cores, then there exists a partition $\{\Gamma_1, \ldots, \Gamma_m\}$ of Γ such that the following condition holds:

$$\forall j \in 1..m, \quad \sum_{\tau_i \in \Gamma_j} U_i(\lambda) \leq 1$$

Consider a value $\lambda' \geq \lambda$. For each task τ_i,

$$U_i(\lambda') = \max\left(U_i^{\max} - \lambda' E_i, U_i^{\min}\right) \leq \max\left(U_i^{\max} - \lambda E_i, U_i^{\min}\right) = U_i(\lambda)$$

Since $U_i(\lambda') < U_i(\lambda)$, it follows that:

$$\forall j \in 1..m, \quad \sum_{\tau_i \in \Gamma_j} U_i(\lambda') \leq \sum_{\tau_i \in \Gamma_j} U_i(\lambda) \leq 1$$

So there remains a partition of $\Gamma(\lambda')$ for which the condition holds. $\square$

It follows that $\Gamma(\lambda)$ is partitioned EDF schedulable for every value of λ that exceeds $\lambda^*_{\Gamma,m}$. This allows us to say something about the optimality of the elastic algorithms for partitioned EDF scheduling.

Theorem 3. *The values of λ obtained by using the iterative approach of Orr and Baruah [118] or the binary search in Algorithm 3 will be within ϵ of λ^* if an exact test of partitioned EDF schedulability is performed for $\Gamma(\lambda)$ at each considered value of λ. In other words, $\lambda - \lambda^* < \epsilon$.*

Proof.

- **Iterative Approach**: The algorithm tests $\lambda = 0$ first; if $\lambda^* = 0$, then the algorithm returns this value. Otherwise, consider the value λ returned by the algorithm: $\Gamma(\lambda)$ is feasible, but $\Gamma(\lambda - \epsilon)$ is *not* feasible. It follows from Theorem 2 that $\lambda^* > \lambda - \epsilon$, which implies $\lambda - \lambda^* < \epsilon$.

- **Binary Search Approach**: The algorithm again tests $\lambda = 0$ first; if $\lambda^* = 0$, then the algorithm returns this value. Otherwise, consider the value λ_{HI} returned by the algorithm: $\Gamma(\lambda_{\text{HI}})$ is feasible, but $\Gamma(\lambda_{\text{LO}})$ is not; thus, by Theorem 2, $\lambda^* > \lambda_{\text{LO}}$. Due to the algorithm's termination condition, we know that $\lambda_{\text{HI}} - \lambda_{\text{LO}} \leq \epsilon$, and so $\lambda - \lambda^* < \epsilon$.

$\square$

Corollary 1. *The values λ_{IT} obtained by the iterative approach of Orr and Baruah [118] and λ_{BS} obtained by Algorithm 3 will be within ϵ of each other if an exact test of partitioned EDF schedulability is performed for $\Gamma(\lambda)$ at each considered value of λ. In other words, $|\lambda_{\text{IT}} - \lambda_{\text{BS}}| < \epsilon$.*

Proof. From Theorem 3, $\lambda_{\text{IT}} - \lambda^* < \epsilon$ and $\lambda_{\text{BS}} - \lambda^* < \epsilon$, so $|\lambda_{\text{IT}} - \lambda_{\text{BS}}| < \epsilon$. $\square$

This tells us that, given an exact schedulability test for partitioned EDF, both algorithms will find values for λ that are within ϵ of the optimal value λ^* and are within ϵ of each other. However, no such guarantee can be made if schedulability is determined by heuristic.

We prove this by example. Consider the set of 32 tasks with parameters listed in Appendix A to be scheduled on 8 cores. When checking schedulability by determining if any of the first fit, worst fit, and best fit heuristics find a feasible partition, the iterative approach of Orr and Baruah [118] returns a value $\lambda_{\text{IT}} = 0.290912449$ for $\epsilon = 0.000538727676$. However, the binary search in Algorithm 3 returns a value $\lambda_{\text{BS}} = 0.300403804$. In this case, $\lambda_{\text{BS}} - \lambda_{\text{IT}} \approx 0.009 > \epsilon$. It follows that:

Theorem 4. *The difference between the values λ_{IT} obtained by the iterative approach of Orr and Baruah [118] and λ_{BS} obtained by Algorithm 3 might exceed ϵ if heuristic tests of EDF schedulability are used.*

This has a surprising implication, which follows from the above results.

Corollary 2. *Given a value of λ, if $\Gamma(\lambda)$ is identified by heuristic to be partitioned EDF schedulable on m cores, then $\Gamma(\lambda')$ might not be identifiable as such for some value of $\lambda' > \lambda$.*

The implication, then, is that while binary search is faster, it might overcompress a set of tasks by more than ϵ when applying heuristic partitioning (of course, the iterative search might overcompress instead). However, as we show in Section 3.5.4, binary search compresses, on average, only $0.262\times\epsilon$ more than iterative search for the sets of tasks we evaluated.

3.4.3 Application of Algorithm 2

In [12], it is observed that under the first-fit and best-fit heuristics, a set Γ of tasks τ_i are schedulable on m processor cores if their total utilization $\sum_i U_i$ does not exceed $(m+1)/2$ and if no single task's utilization exceeds 1. Thus, the efficient procedure outlined in Algorithm 2 can be adopted by compressing to a desired utilization U_D=$(m+1)/2$, achieving compression in $\boldsymbol{\mathcal{O}(n \log n)}$ time.

We note that $(m+1)/2$ is an *upper-bound* on the utilization required by the first-fit and best-fit heuristics. Thus, the amount of compression resulting from an application of this approach might be more than necessary to achieve partitioned EDF schedulability, even under the above-listed heuristics. It follows that the approach of Orr and Baruah [118], while slower, might achieve better results — both in terms of compressing utilizations less aggressively, and by identifying more schedulable task sets. We evaluate these tradeoffs in Section 3.5.4.

3.5 Evaluation

We now compare our proposed approaches in Section 3.4 to elastic partitioned EDF scheduling of implicit-deadline tasks to the original approach proposed by Baruah and Orr [118].

3.5.1 Implementation

Evaluations are performed in the same fashion on the same Raspberry Pi 3 Model B+ as described in Section 2.4.1. Tasks τ_i are also represented using the same data structure.

We compare the following five implementations:

1. ITER: The iterative approach from [118]. For each value of λ tested, heuristics to check schedulability are applied in the order specified in [118]: (1) first fit, (2) worst fit, then (3) best fit. The algorithm terminates once any of these deems the task system schedulable.

2. ITER-ORDER: The same iterative approach, but heuristics are applied according to their ability to find a feasible partition; this means that the algorithm is likely to terminate earlier. We determine this order empirically.

3. BS: Our proposed binary search approach in Algorithm 3. Heuristics are applied in the same order specified in [118].

4. BS-ORDER: The same binary search approach, but heuristics applied in the empirically-determined best order.

5. UTIL: The utilization-based approach of Algorithm 2, with $U_D = (m+1)/2$. We use the array-based implementation, as this was observed to perform best in our evaluations in Section 2.4.3.

3.5.2 Generating Task Sets

We generate sets Γ of tasks τ_i according to Orr and Baruah's methodology in [118]:

- We consider multiprocessor platforms with $m = 4$, 8, and 16 identical processor cores.
- For each number of cores m, we consider sets of n tasks, with $n = 2m$, $4m$, and $8m$.
- The maximum utilization $U_i^{\max}$ assigned to each task τ_i is selected at random, but we constrain these values to be no more than a parameter α. We separately consider values of $\alpha \in \{0.6, 0.8, 1.0\}$.
- Each set of tasks has a total maximum utilization $U_{\text{SUM}}^{\max}$ of $u{\cdot}m{\cdot}\alpha$. We separately consider values of $u \in \{1.1, 1.5, 1.9\}$.
- For each combination of values m, n, α, and u, we generate 1000 sets of tasks.

- We use the DRS algorithm [70] to distribute the total maximum utilization $U_{\text{SUM}}^{\max}$ across individual $U_i^{\max}$ values. DRS allows us to select these values uniformly from the space of selections satisfying the conditions that *(i)* the total $\sum_i U_i^{\max}$ equals the specified $U_{\text{SUM}}^{\max}$ and *(ii)* each value $U_i^{\max}$ does not exceed the constraint imposed by the chosen value of α.

- Individual minimum utilizations $U_i^{\min}$ are assigned at random, selected uniformly from the range $(0, U_i^{\max}]$.

- Elastic coefficients E_i are assigned at random, selected uniformly from the range $(1, 5]$.

For each set of tasks, we compute $\lambda^{\max}$ per Equation 3.2. For all implementations of the algorithm in [118], we use the same "granularity" $\epsilon = \frac{\lambda^{\max}}{1000}$ as Orr and Baruah.

3.5.3 Determining a Heuristic Order

For each task set, we run implementation Iter to find the minimum value of λ for which at least one of the three considered partitioning heuristics determines feasibility. Then for each heuristic, we count how many of the task sets Γ it determines to be feasible at $\Gamma(\lambda)$. Results are summarized in Table 3.1.

Heuristic	Count	Percentage of Total
First Fit	77 041	95.1%
Worst Fit	14 543	18.0%
Best Fit	78 645	97.1%
Total	81 000	100%

Table 3.1: Number of task sets determined feasible by each heuristic.

We observe that, of the 135 000 compressed task sets $\Gamma(\lambda)$ considered, ***the best fit heuristic identifies a feasible partition at the highest rate***, while first fit does almost as well; therefore, we implement the Iter-Order and BS-Order approaches to first attempt the best fit, followed by the first fit. The worst fit heuristic does the worst by far; furthermore, there are no task sets for which the worst fit heuristic identifies a feasible task set that is not also identified by both the best and first fit. As such, ***we eliminate the worst fit heuristic*** from both implementations.

3.5.4 Comparison of Improvements

For each set Γ of tasks τ_i, we note for each implementation *(i)* its execution time in CPU cycles, *(ii)* whether it deems Γ to be schedulable, and if so, *(iii)* the amount of compression λ it requires. This allows us to answer the following questions.

How much is gained by changing heuristics?

We begin by comparing the execution times of the ITER and BS implementations to those of the ITER-ORDER and BS-ORDER implementations. Results are illustrated in Figures 3.1–3.4, which show — for each combination of m, α, n, and u — the median and maximum execution times in processor cycles. Figure 3.1 compares the median execution times of ITER and ITER-ORDER, while Figure 3.2 compares the maximum execution times. Figures 3.3 and 3.4 compare the median and maximum execution times for BS and BS-ORDER.

We observe that by applying best fit bin packing, then first fit, and by not applying worst fit bin packing, ***the time to heuristically assess partitioned EDF schedulability is reduced dramatically.*** The median execution time of ITER-ORDER is up to 2.40× faster than ITER, and the maximum execution time is up to 2.41× faster. The median execution time of BS-ORDER is up to 1.76× faster than BS, and the maximum execution time is up to 2.23× faster.

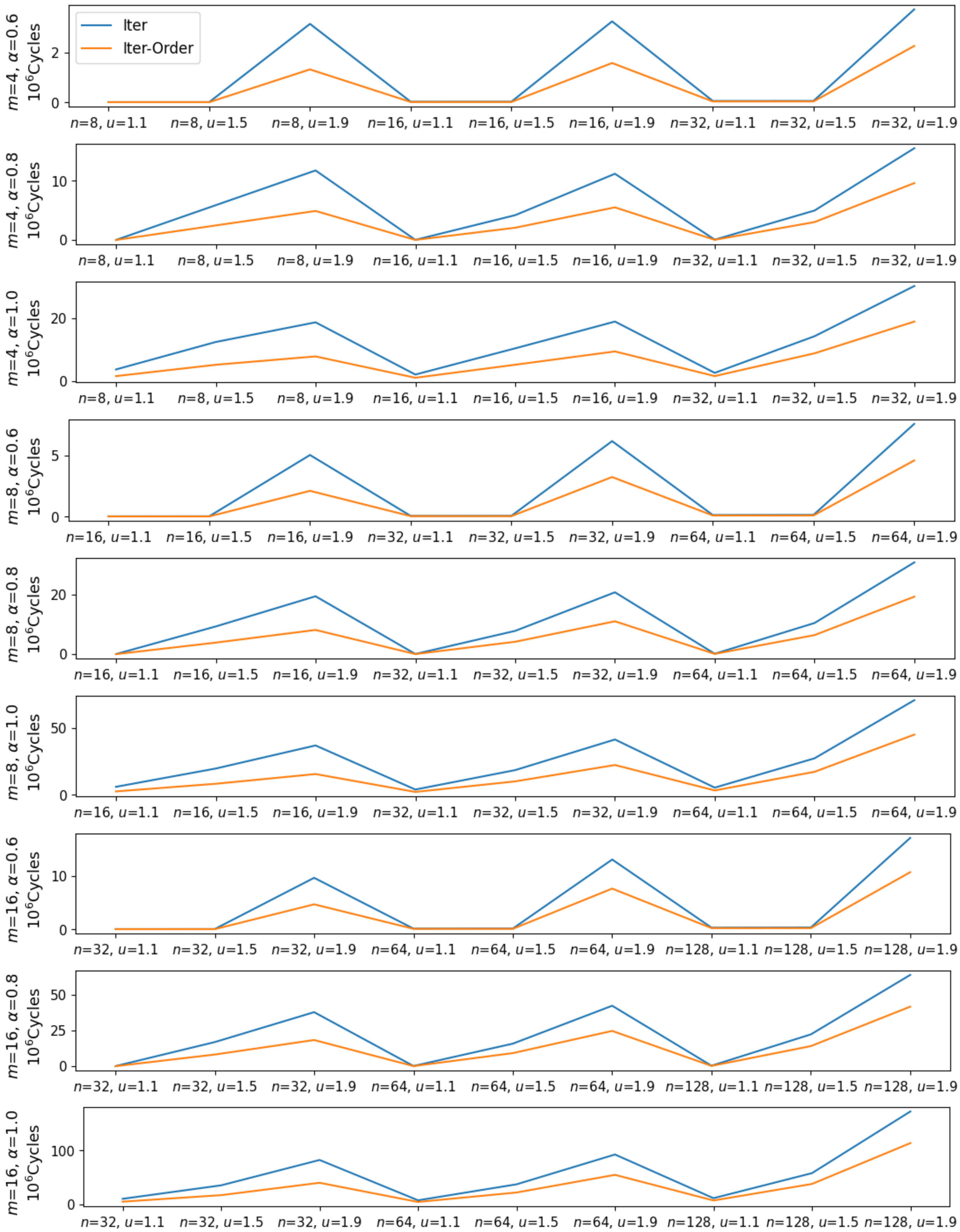


Figure 3.1: Median execution times for Iter and Iter-Order in CPU Cycles.

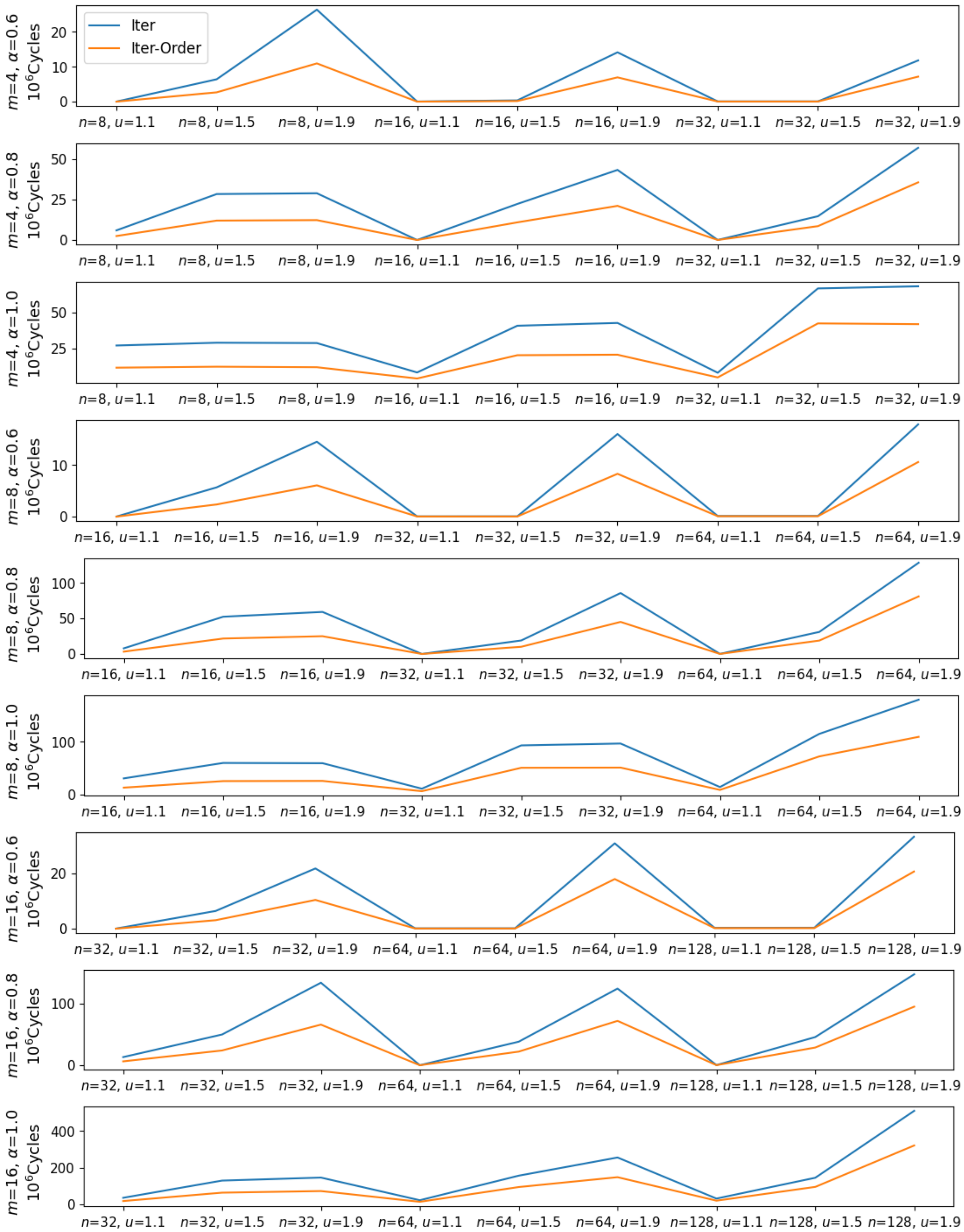


Figure 3.2: Maximum execution times for ITER and ITER-ORDER in CPU Cycles.

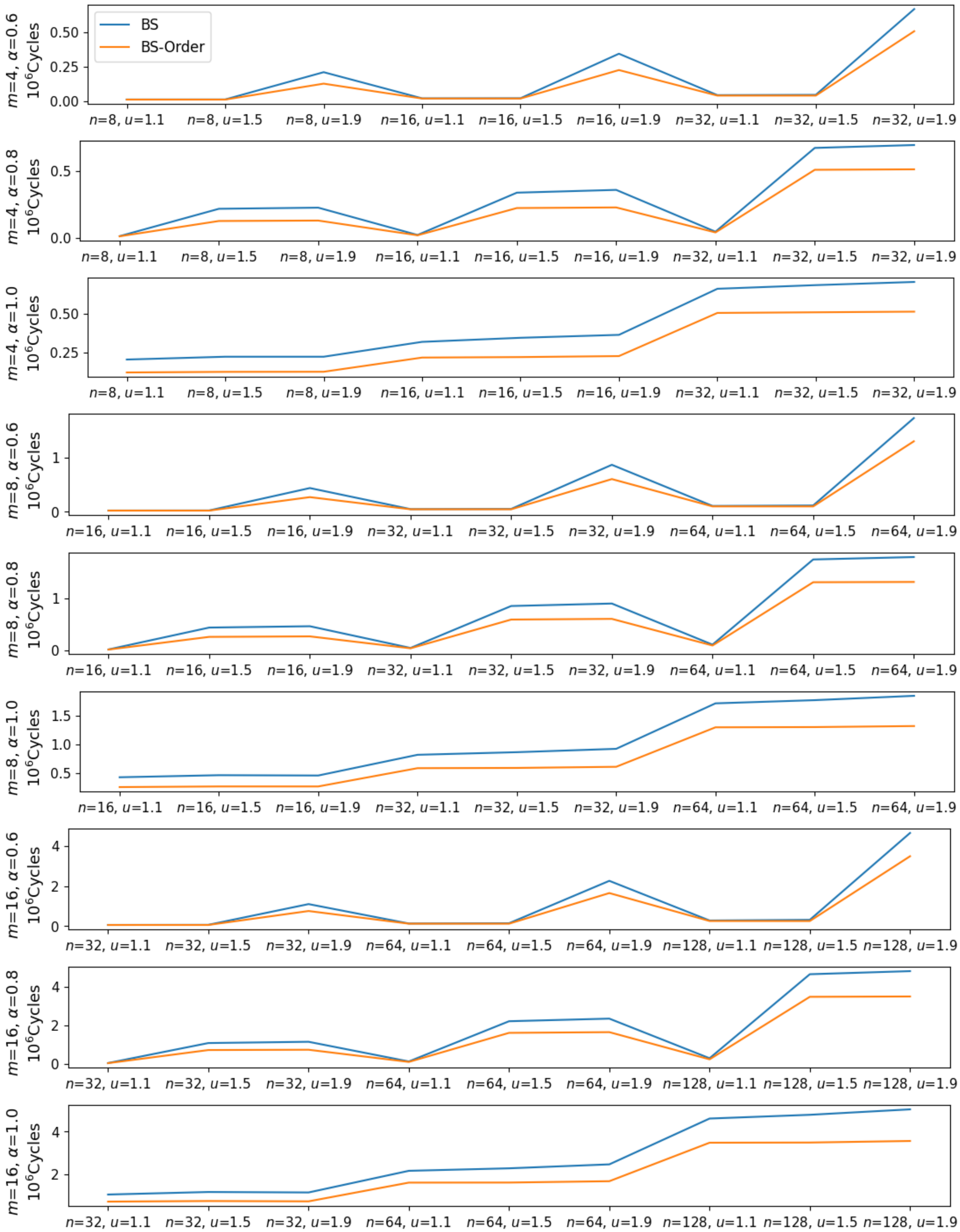


Figure 3.3: Median execution times for BS and BS-Order in CPU Cycles.

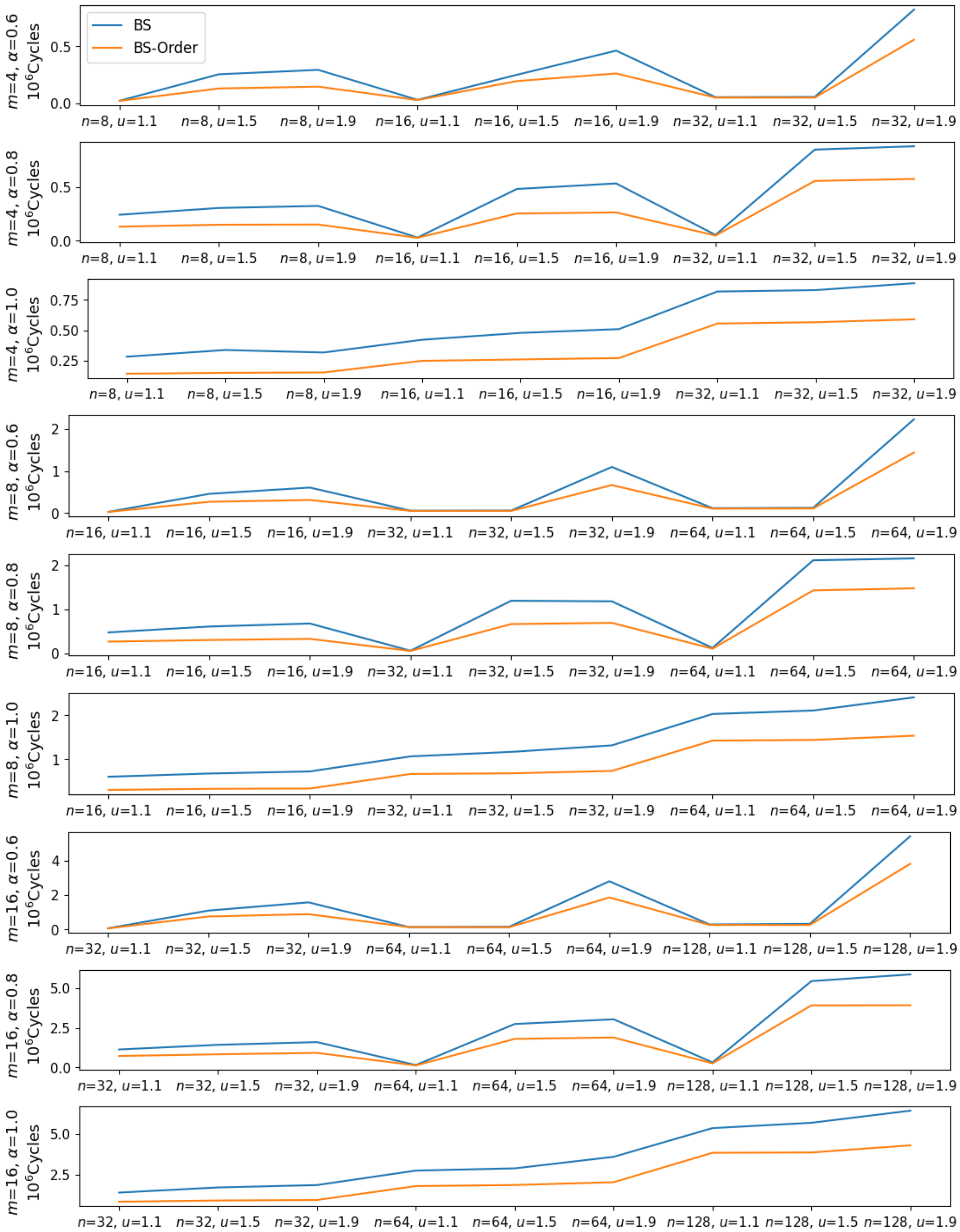


Figure 3.4: Maximum execution times for BS and BS-Order in CPU Cycles.

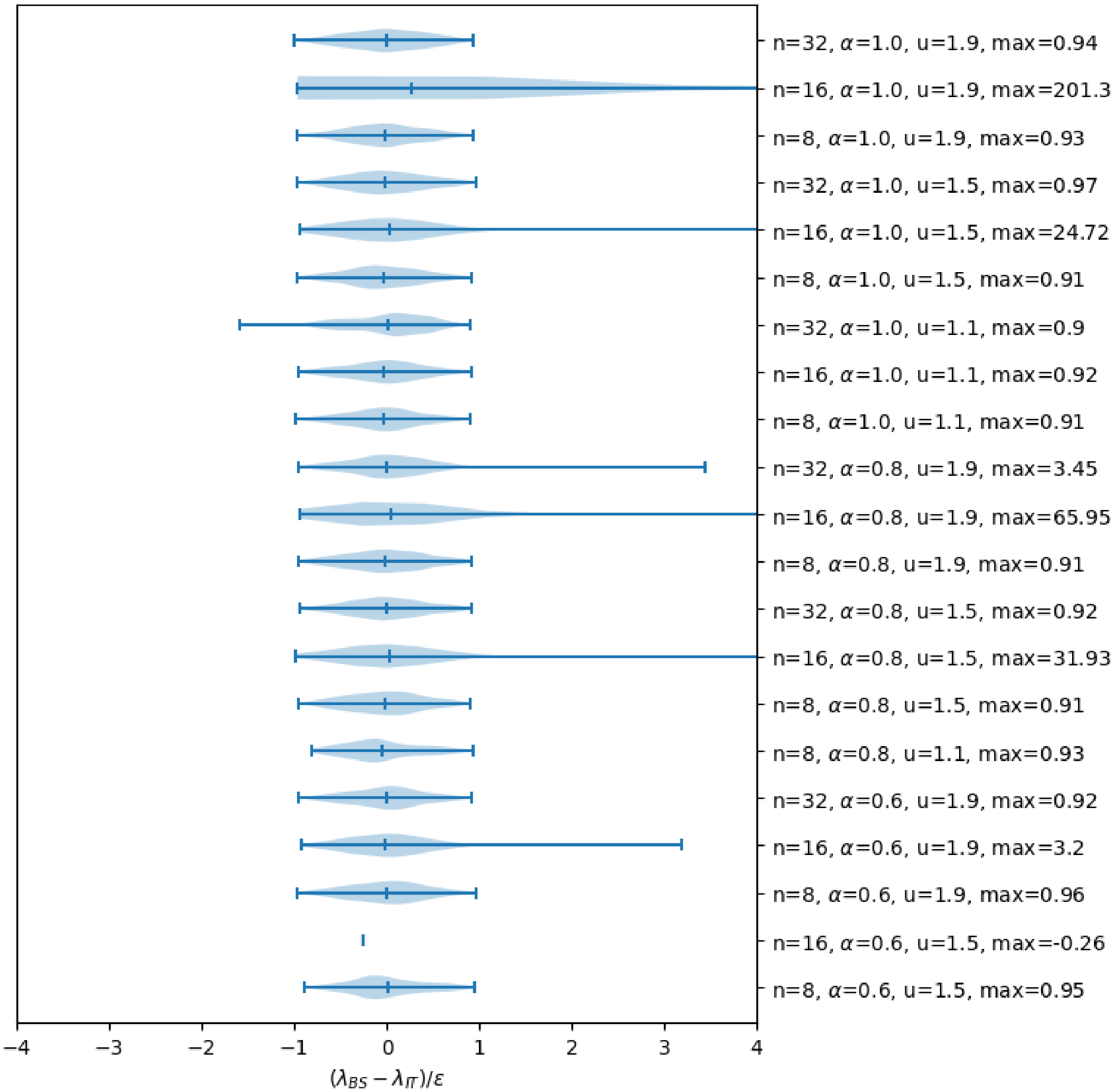
Does performing binary search affect the amount of compression applied?

Per Theorem 4, the difference between the λ values returned by implementations ITER and BS might exceed ϵ. We quantify this, and consider whether one implementation consistently compresses more than the other.[9]

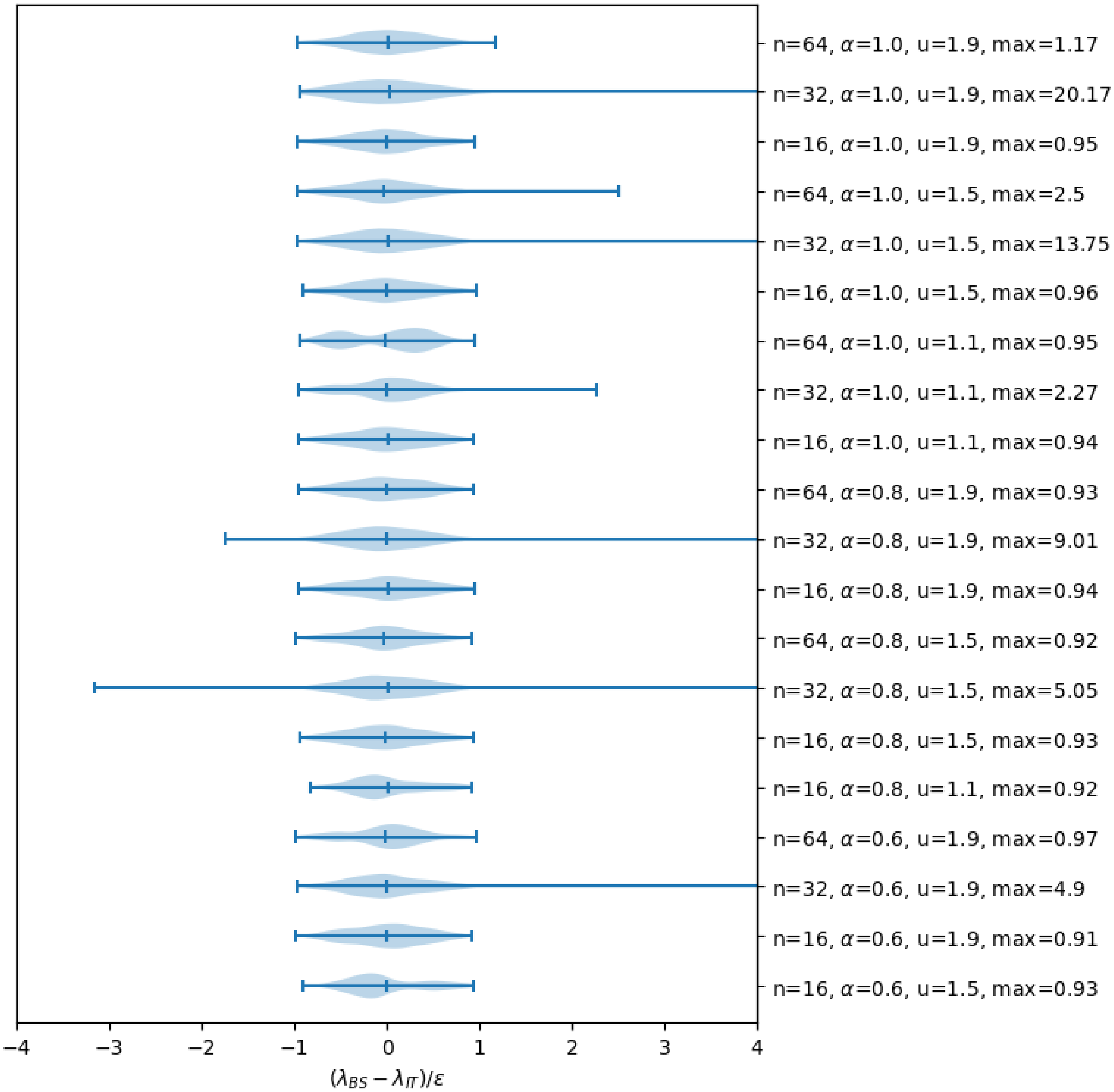
Results are illustrated in Figure 3.5, which shows — for each combination of m, n, α, and u for which any task sets required compression — the distribution of differences $(\lambda_{\text{BS}} - \lambda_{\text{IT}})/\epsilon$ between the value λ_{BS} returned by BS and λ_{IT} returned by ITER, normalized by ϵ. Task sets not requiring compression ($\lambda = 0$) are not included, nor are those not determined to be schedulable under any amount of compression. Where outliers extend beyond the plotted boundaries, the y-axis labels denote the maximum value.

We observe that, although the values λ_{BS} and λ_{IT} typically do not differ by more than ϵ, there are cases where they differ by much more. For 16 tasks on 4 cores, with $\alpha = 1.0$ and $u = 1.9$, λ_{BS} exceeds λ_{IT} by more than $200\times\epsilon$. Generally, we see that outliers occur where λ_{BS} is larger than λ_{IT}. This makes sense due to the behavior of binary search: search proceeds downward in factors of 2 from larger values, and if $\Gamma(\lambda)$ is deemed unschedulable for some tested value of λ greater than the optimal λ^*, the binary search will continue to test larger values.

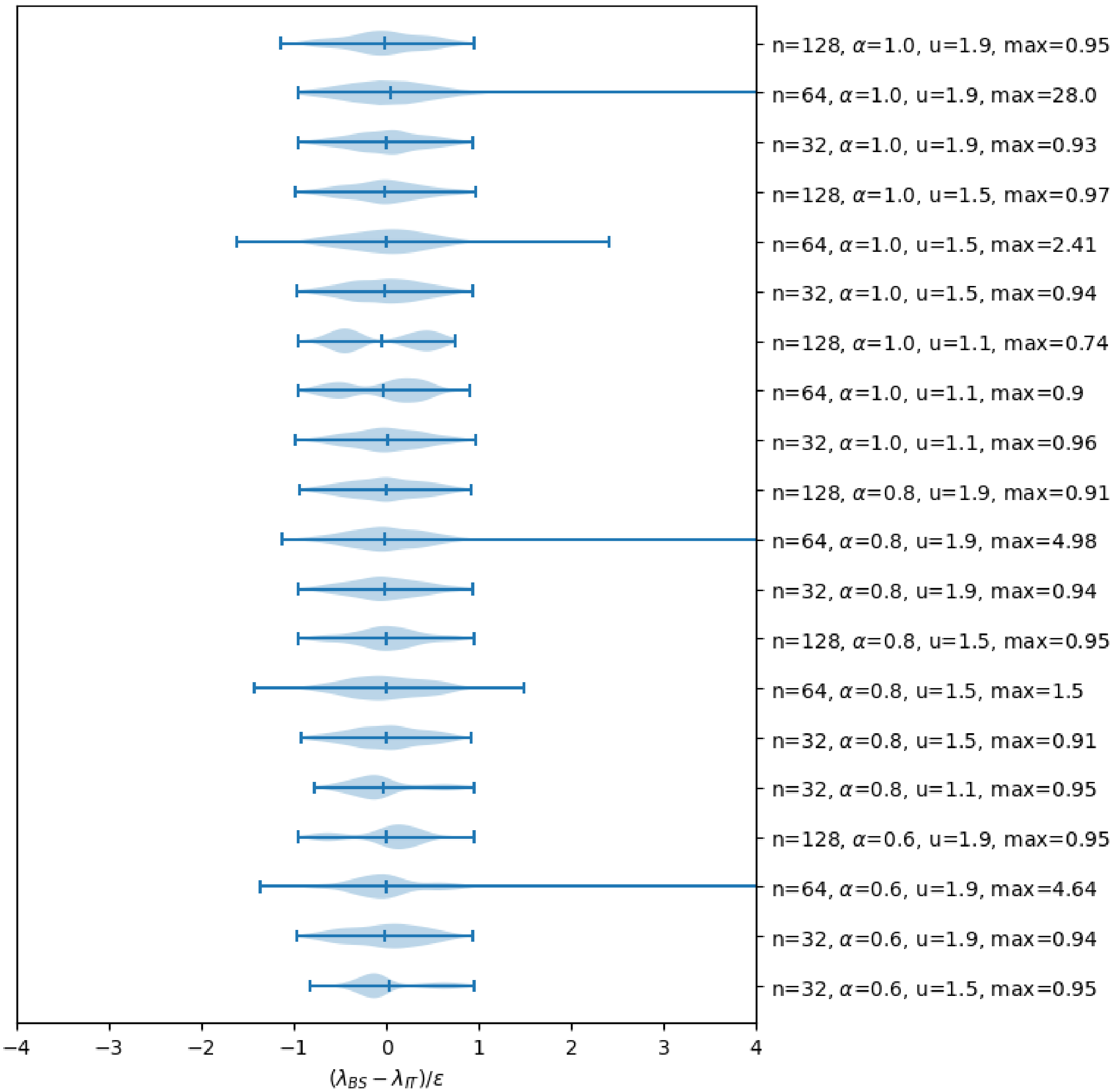
[9] The ITER-ORDER implementation will always return the same value for λ as ITER; similarly for BS-ORDER and BS, so we do not separately consider these here.


n=32, α=1.0, u=1.9, max=0.94
n=16, α=1.0, u=1.9, max=201.3
n=8, α=1.0, u=1.9, max=0.93
n=32, α=1.0, u=1.5, max=0.97
n=16, α=1.0, u=1.5, max=24.72
n=8, α=1.0, u=1.5, max=0.91
n=32, α=1.0, u=1.1, max=0.9
n=16, α=1.0, u=1.1, max=0.92
n=8, α=1.0, u=1.1, max=0.91
n=32, α=0.8, u=1.9, max=3.45
n=16, α=0.8, u=1.9, max=65.95
n=8, α=0.8, u=1.9, max=0.91
n=32, α=0.8, u=1.5, max=0.92
n=16, α=0.8, u=1.5, max=31.93
n=8, α=0.8, u=1.5, max=0.91
n=8, α=0.8, u=1.1, max=0.93
n=32, α=0.6, u=1.9, max=0.92
n=16, α=0.6, u=1.9, max=3.2
n=8, α=0.6, u=1.9, max=0.96
n=16, α=0.6, u=1.5, max=-0.26
n=8, α=0.6, u=1.5, max=0.95
−4 −3 −2 −1 0 1 2 3 4
$(\lambda_{BS} - \lambda_{IT})/\varepsilon$

(a) $m = 4$ Cores.


n=64, α=1.0, u=1.9, max=1.17
n=32, α=1.0, u=1.9, max=20.17
n=16, α=1.0, u=1.9, max=0.95
n=64, α=1.0, u=1.5, max=2.5
n=32, α=1.0, u=1.5, max=13.75
n=16, α=1.0, u=1.5, max=0.96
n=64, α=1.0, u=1.1, max=0.95
n=32, α=1.0, u=1.1, max=2.27
n=16, α=1.0, u=1.1, max=0.94
n=64, α=0.8, u=1.9, max=0.93
n=32, α=0.8, u=1.9, max=9.01
n=16, α=0.8, u=1.9, max=0.94
n=64, α=0.8, u=1.5, max=0.92
n=32, α=0.8, u=1.5, max=5.05
n=16, α=0.8, u=1.5, max=0.93
n=16, α=0.8, u=1.1, max=0.92
n=64, α=0.6, u=1.9, max=0.97
n=32, α=0.6, u=1.9, max=4.9
n=16, α=0.6, u=1.9, max=0.91
n=16, α=0.6, u=1.5, max=0.93
−4
−3
−2
−1
0
1
2
3
4
(λBS − λIT)/ε

(b) $m = 8$ Cores.



(c) $m = 16$ Cores.

Figure 3.5: Difference between λ_{BS} and λ_{IT}, normalized by ϵ.

What are the tradeoffs between execution time and amount of compression required?

To further evaluate the implications of the observed outliers in $(\lambda_{\text{BS}} - \lambda_{\text{IT}})/\epsilon$ on the average-case compression of each algorithm, and to consider what is otherwise gained in speedup, we perform a side-by-side comparison of the execution time and achieved λ values for our ITER-ORDER and BS-ORDER implementations. Because we have already shown that these implementations are significantly faster than their respective ITER and BS counterparts, we remove the latter from further comparison.

As before, we only consider those task sets that are schedulable with compression ($\lambda > 0$). For each combination of m, α, n, and u, Figure 3.6 illustrates the median and maximum speedup achieved by BS-ORDER over ITER-ORDER. We do not plot mean values of λ produced by each implementation, as ***their average compression values agree closely***, differing by less than $0.262\times\epsilon$ for every considered combination.

We also ***observe that*** BS-ORDER ***achieves significant speedups***, especially for larger values of α and u. These task sets have larger total maximum utilizations $U_{\text{SUM}}^{\max}$, and therefore tend to need more compression to achieve schedulability. In such cases, the iterative approach takes longer to reach the higher λ value, so the binary search is significantly faster. The median speedups for each combination were as high as $51\times$, while the maximum speedup observed was $86\times$.

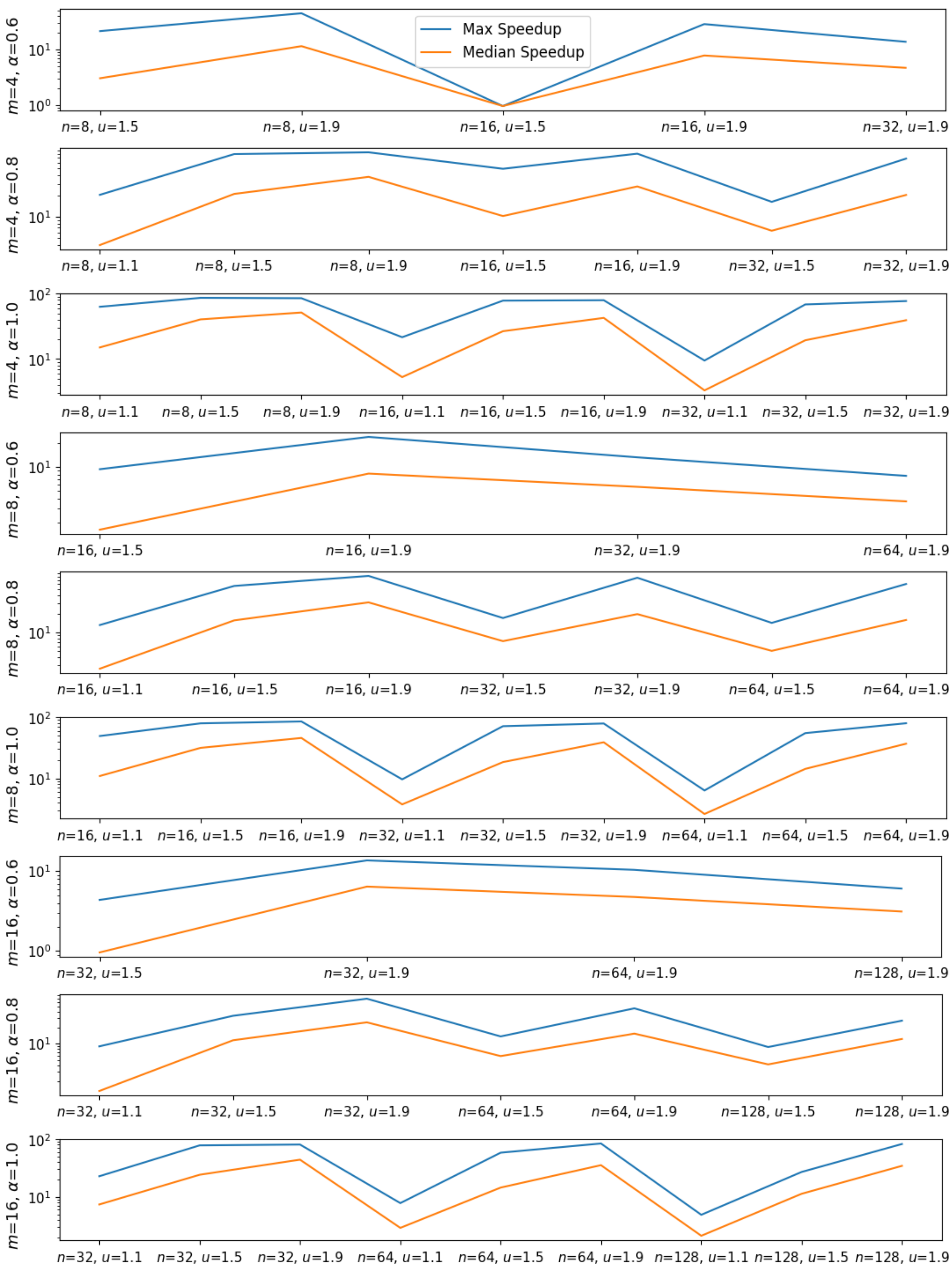


Figure 3.6: Speedups achieved by BS-Order over Iter-Order.

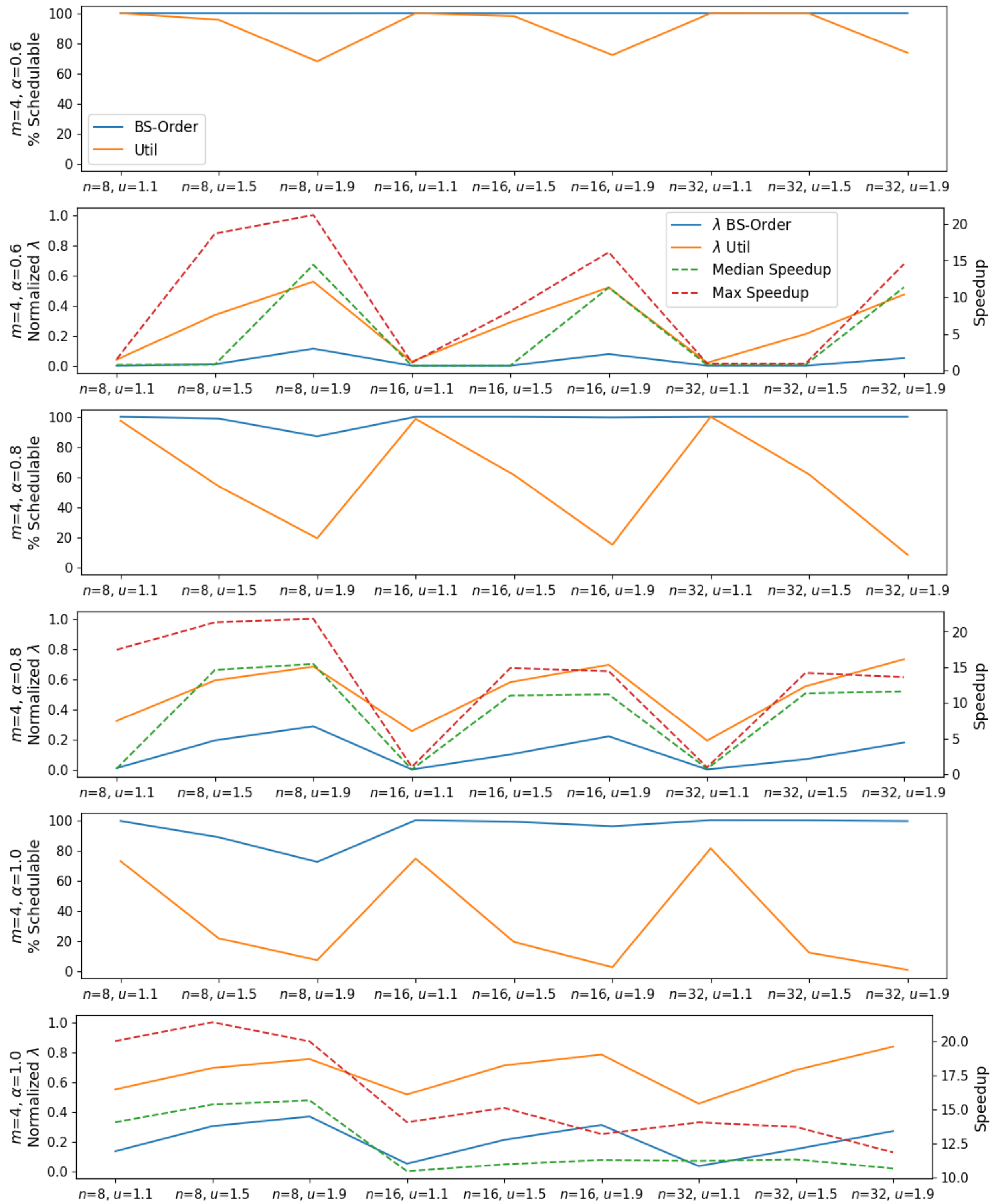
How well does our application of Algorithm 2 perform?

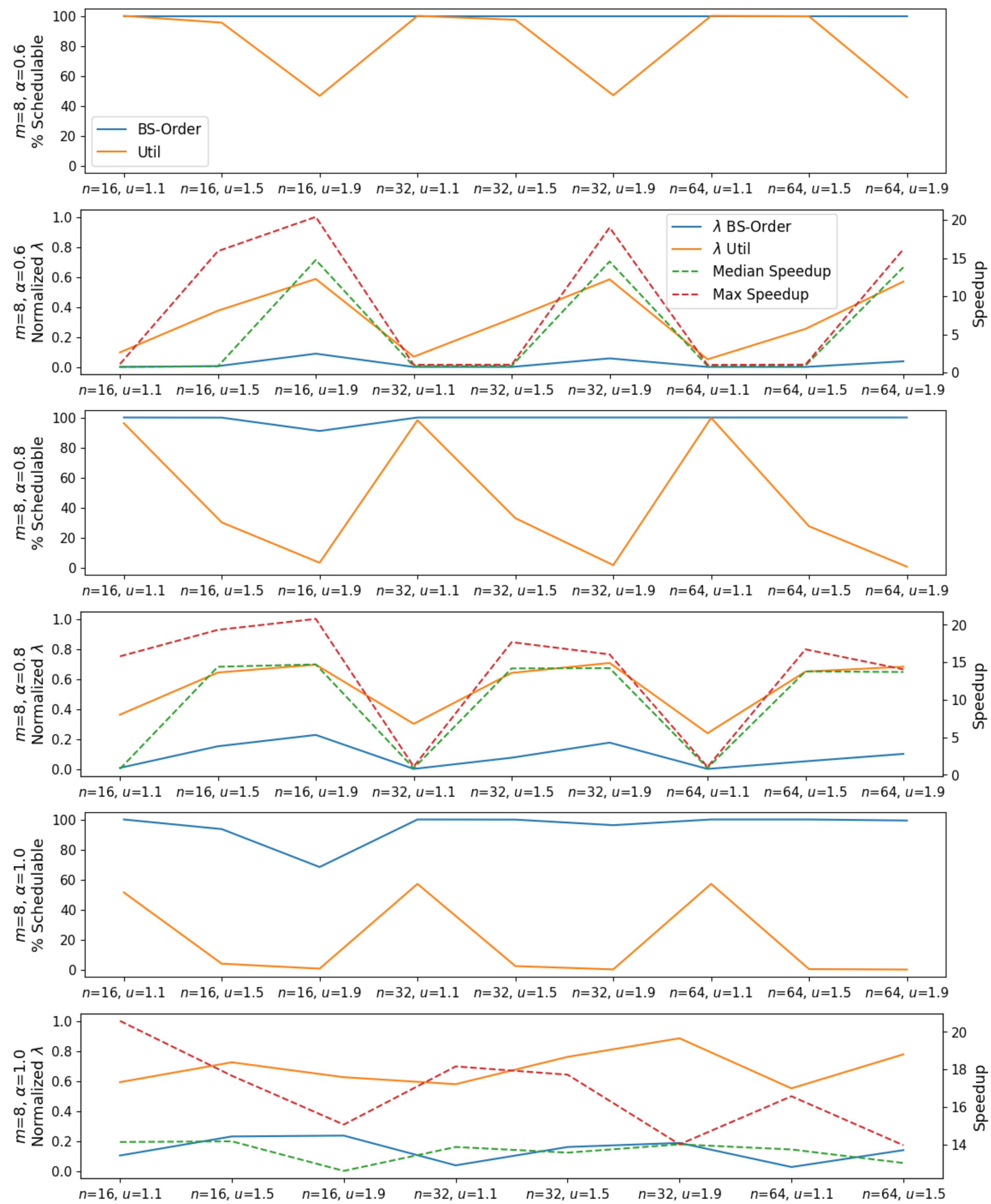
To this point, we have determined that BS-Order performs significantly faster than the other implementations that search for a value of λ with granularity ϵ, while having minimal impact on the quality of the solution. We therefore select it for comparison with our Util implementation. Recall that Util applies our quasilinear procedure, outlined in Algorithm 2, using the utilization bound of $U_D = (m+1)/2$ achievable by the best fit and first fit heuristics on m cores [12]. While we expect it to perform faster, we also expect it to be more pessimistic.

We evaluate this hypothesis by comparing the rates at which BS-Order and Util determine schedulability, the resulting values of λ necessary to achieve schedulability, and the time to find those values of λ. As in [118], to ensure a consistent comparison, we only compare λ values (and, in our case, execution times — measuring execution times was outside the scope of the work in [118]) for those tasks deemed schedulable. Also as in [118], we separately consider each value of m, α, n, and u.

Figure 3.7 contains plots that alternate between two types. The first type shows the percentage of schedulable task sets identified by BS-Order and Util for every combination of m, α, n, and u. The second type compares the median and maximum execution time speedup gained by Util over BS-Order to the mean λ values achieved by each implementation. As in [118], λ values are normalized by $\lambda^{\max}$ to give a value in the interval [0,1]; this is necessary for comparing λ values across task sets.

We observe that, while ***significant speedups are achieved*** by Util over BS-Order, Util identifies fewer schedulable task sets, and for those it does identify as schedulable, it typically imposes more compression. This is especially true for larger values of α and u for which the total maximum utilization $U_{\text{SUM}}^{\max}$ is larger. Nonetheless, at the cost of more pessimism, Util achieves speedups observed to reach over $20\times$.


m=4, α=0.6
% Schedulable
BS-Order
Util
m=4, α=0.6
Normalized λ
λ BS-Order
λ Util
Median Speedup
Max Speedup
Speedup
m=4, α=0.8
% Schedulable
m=4, α=0.8
Normalized λ
Speedup
m=4, α=1.0
% Schedulable
m=4, α=1.0
Normalized λ
Speedup
n=8, u=1.1
n=8, u=1.5
n=8, u=1.9
n=16, u=1.1
n=16, u=1.5
n=16, u=1.9
n=32, u=1.1
n=32, u=1.5
n=32, u=1.9



m=8, α=0.6
% Schedulable
BS-Order
Util
n=16, u=1.1
n=16, u=1.5
n=16, u=1.9
n=32, u=1.1
n=32, u=1.5
n=32, u=1.9
n=64, u=1.1
n=64, u=1.5
n=64, u=1.9
m=8, α=0.6
Normalized λ
λ BS-Order
λ Util
Median Speedup
Max Speedup
Speedup
m=8, α=0.8
% Schedulable
m=8, α=0.8
Normalized λ
Speedup
m=8, α=1.0
% Schedulable
m=8, α=1.0
Normalized λ
Speedup

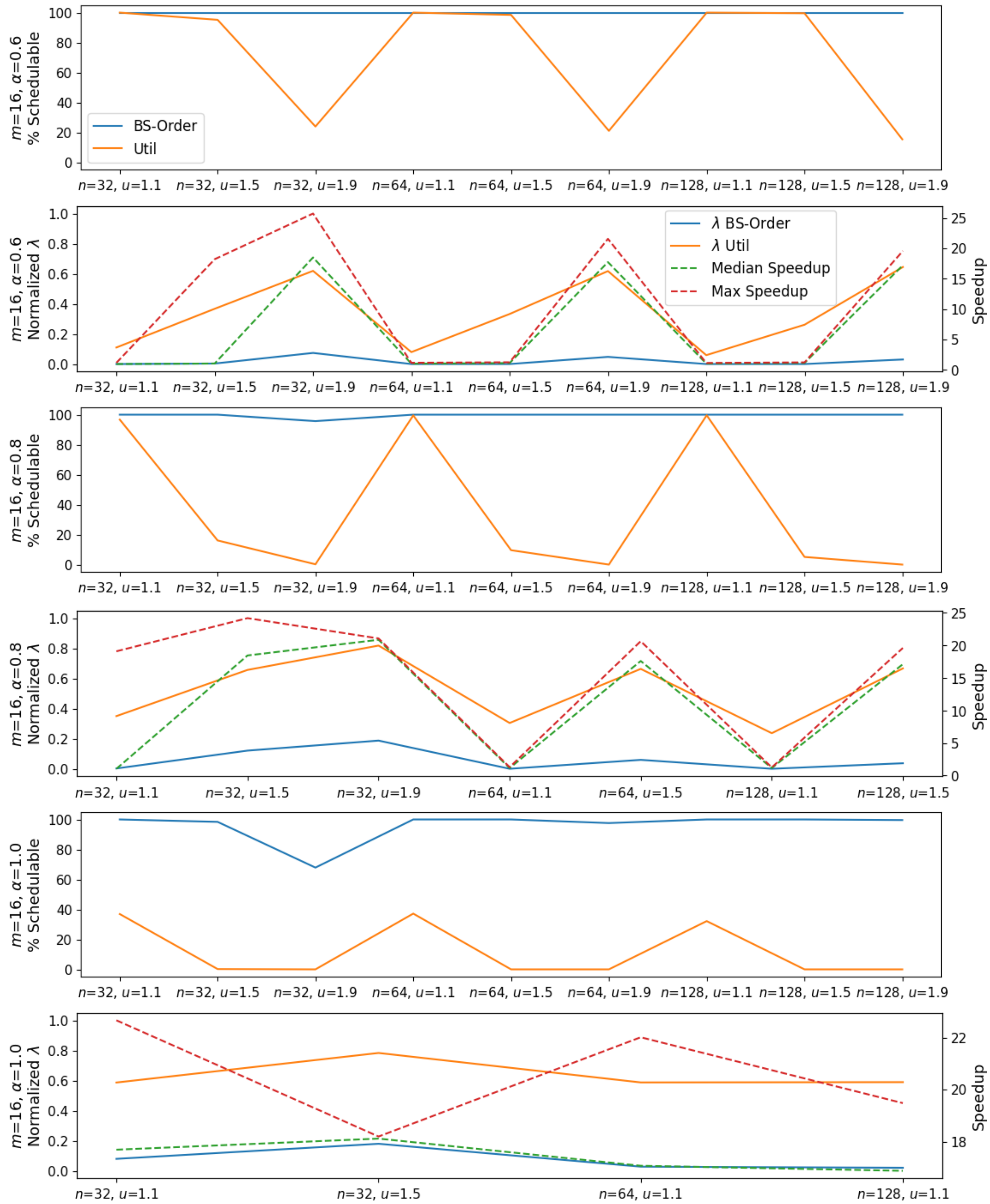


Figure 3.7: Speed and Schedulability Tradeoffs Between BS-Order and Util.

3.6 Conclusion

In this chapter, we have considered alternative approaches toward extending elastic scheduling to the fluid and partitioned EDF multiprocessor implicit-deadline models, comparing them to the original approaches proposed by Orr and Baruah in [118]. We have evaluated their execution time, and compared to their ability to avoid pessimistic overcompression.

In Section 2.4, we demonstrated that Buttazzo's original algorithm [37, Figure 9.29] for elastic scheduling [39, 40] is significantly slower to compress tasks compared to our proposed approach in Algorithm 2. In this chapter, we discussed how the same results apply to fluid scheduling. This suggests that ***our improved algorithm should be used with fluid scheduling to handle admission control or to respond to changes in the number of available processor cores.***

While fluid scheduling is a convenient abstraction, it is difficult to implement in practice. We therefore also considered partitioned EDF scheduling. We demonstrated that simple modifications to the approach in [118] — ***changing which partitioning heuristics are applied***, and ***performing a binary search*** instead of iterating over the space of compression to apply — yield substantial improvements in execution time without significantly overcompressing.

We also demonstrated an application of our Algorithm 2 to partitioned EDF elastic scheduling. While it runs even faster, it is often pessimistic, at times overcompressing or deeming a schedulable task set to be infeasible. Nonetheless, we can imagine scenarios where this may be desirable. For example, in mixed-criticality systems [161, 32], if a job of a safety-critical task overruns its expected execution time, jobs of less critical tasks are traditionally dropped. Elastic frameworks have been proposed instead where the periods of low-criticality tasks are extended to maintain the service-level guarantees required by critical jobs [142]. In such a situation, the decision must be made as quickly as possible. If our faster approach overcompresses, this is no worse than the inelastic case where all low-criticality jobs are dropped anyway.

Chapter 4

Constrained-Deadline Tasks

Portions of this chapter were published as "Elastic Scheduling for Fixed-Priority Constrained Deadline Tasks" at ISORC 2023, winning the best paper award [143].

4.1 Introduction

In the previous chapter, we considered elastic scheduling of implicit-deadline tasks. This chapter extends the elastic model to fixed-priority constrained-deadline task systems on a uniprocessor. In particular, we consider tasks with adaptable rates — their utilizations can be compressed by extending their periods — but with *deadlines* that are held *constant*.

Prior work has considered such tasks when scheduled in EDF fashion. In [44, 45], Chantem et al. demonstrated the equivalence of elastic scheduling to the problem of minimizing the weighted sum of squared deviations of each task's compressed utilization from its nominal value, constrained by the tasks' minimum utilizations and the utilization bound of the system. They used this result to extend elastic scheduling to constrained-deadline tasks by replacing the constraint on total utilization with a tractable approximation of the processor-demand analysis (PDA) [20] test for earliest-deadline first (EDF) schedulability.

More recently, Baruah [13] demonstrated that these approximations result in a high degree of pessimism for certain task sets. Instead, Baruah presented an alternative that uses an iterative approach — similar in spirit to that of Orr and Baruah [118] for partitioned EDF scheduling discussed in Section 3.2.2 — that increases total compression in constant-size steps until the system is schedulable according to PDA. The algorithm is tuned by selecting the iteration granularity; a smaller step increases the running time of the algorithm but allows for a more precise result.

In this chapter, we extend Baruah's approach in [13] to ***fixed-priority*** uniprocessor scheduling of systems of constrained-deadline elastic tasks. We first present an iterative algorithm that similarly increases compression until schedulability is achieved according to the response time analysis (RTA) test [6]. We then present two refinements to this algorithm, both leveraging the observation that once a task has been compressed to schedulability according to RTA, it remains schedulable when more compression is applied to the system. The first refinement iterates over tasks in order of decreasing priority, increasing total compression until that task is schedulable under RTA before considering the next task. The second performs binary search over the range of allowed compression, skipping RTA for tasks that are already known to be schedulable at lower levels of compression.

Next, we formulate the problem of finding the optimal amount of compression to guarantee schedulability as a mixed integer quadratic program (MIQP). Due to the large number of quadratic constraints, the problem may be difficult to solve efficiently. However, we can instead reformulate the problem for a *single* task. This enables an alternative algorithm that considers each task in turn; if RTA deems a task unschedulable for the current level of system compression, the MIQP is solved to find the exact amount of compression necessary to schedule that task. By iterating over each task, we can determine the minimum sufficient compression that must be applied to the task system.

We implement the latter four approaches, using the free and open-source SCIP [2] constraint integer programming tool to solve the MIQP. By evaluating each algorithm for randomly-generated synthetic task sets, we demonstrate that the approximate procedures are both highly efficient and typically give a result close enough to optimal to be useful for online scheduling decisions in low-powered embedded devices. We also show that, when an optimal solution is desired, the MIQP-based algorithm may be feasibly solved offline to compress task periods.

The remainder of this chapter is organized as follows:

- Section 4.2 provides the necessary background on system models used in this chapter.
- Section 4.3 presents a basic iterative algorithm to compress tasks until RTA guarantees schedulability.
- Sections 4.4 and 4.5 refine the algorithm to a more efficient iterative approach and a binary search, respectively.

- Section 4.6 formulates an MIQP representation of the problem of finding the minimum amount of compression necessary to schedule the task system.
- Section 4.7 reformulates the problem for a single task, then demonstrates an algorithm to apply this to the complete task system.
- In Section 4.8, we show the results of our evaluation of those approaches.
- Finally, Section 4.9 concludes the chapter and discusses the contexts under which each approach may be relevant.

4.2 Background and System Model

Coverage of the elastic model for recurrent, implicit-deadline tasks on a uniprocessor [39, 40] can be found in Section 2.2.2. In this section, we introduce necessary background on elastic scheduling for constrained-deadline tasks.

4.2.1 Elastic Scheduling for Constrained-Deadline Tasks

In [44, 45], Chantem et al. showed that utilizations selected by the elastic model also solve the following quadratic programming problem:

$$\min_{U_i} \quad \sum_{i=1}^{n} \frac{1}{E_i}(U_i^{\max} - U_i)^2 \tag{4.1a}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} U_i \leq U_D \tag{4.1b}$$

$$\forall_i, \quad U_i^{\min} \leq U_i \leq U_i^{\max} \tag{4.1c}$$

This allowed for an extension of the model to constrained-deadline tasks. A task $\tau_i = (C_i, D_i, U_i^{\min}, U_i^{\max}, U_i, E_i)$ is now characterized with an additional parameter, D_i, representing a relative deadline that remains fixed even if the task's period is extended in response to reduced utilization. Under the constrained deadline model, only tasks for which $D_i \leq T_i$ (i.e., $D_i \leq C_i/U_i^{\max}$) are considered.

The schedulability constraint (Expression 4.1b) is replaced by a representation of the PDA [20] schedulability test. PDA is an optimal technique for schedulability analysis of constrained-deadline sporadic task systems under preemptive EDF scheduling on a uniprocessor. It considers the demand bound function $\textsc{dbf}_i(t)$ for each task τ_i, which denotes the maximum possible cumulative execution required by jobs of the task that arrive and have their deadlines within any contiguous interval of duration $t \geq 0$. This can be computed as:

$$\textsc{dbf}_i(t) = \max\left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1, 0\right) \times C_i \tag{4.2}$$

For $t \geq 0$ and $D_i \leq T_i$, it can be expressed more simply as:

$$\textsc{dbf}_i(t) = \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) \times C_i \tag{4.3}$$

A constrained-deadline task system $\Gamma = \{\tau_1, \ldots, \tau_n\}$ is schedulable under preemptive EDF on a uniprocessor if and only if for all $t > 0$,

$$\sum_{i=1}^{n} \textsc{dbf}_i(t) \leq t \tag{4.4}$$

In [20], it was shown that it is sufficient to check this condition for values of t within the first hyperperiod that take the form $(k \cdot T_i + D_i)$ for non-negative integers k. This set of values constitute the PDA *testing set*. For elastic scheduling, Chantem et al. [44, 45] add a constraint in the form of Expression 4.4 for each element of the testing set to the optimization problem represented by Expression 4.1.

However, the resulting problem might not be tractable. The size of the testing set may be exponential in general, and pseudo-polynomial for bounded-utilization tasks, resulting in an optimization problem with too many constraints to be efficiently solvable. Also, due to its use of the floor function, $\textsc{dbf}_i(t)$ is not a linear expression, and so the optimization problem does not remain a quadratic program. Chantem et al. [44, 45] over-approximate the demand-bound function by removing the floor. Nonetheless, because the test set itself depends on the task periods, the times defining the RHS of the constraints formed by Expression 4.4 are variables in the optimization problem, and the problem remains non-linear. They therefore introduce an approximate form of the problem, and a heuristic approach to solving it, and proved both correct in the sense that the resulting compressed system would be guaranteed schedulable.

4.2.2 Improved Elastic Scheduling for Constrained-Deadline EDF

In recent work [13], Baruah showed that these approximations are highly conservative and may result in significant overcompression for certain task sets. Two alternative approaches were presented; in this paper, we extend these to fixed-priority DM scheduling. Baruah again used the term λ to represent the degree by which compression is applied to the task system. Recall from Equation 3.1 that the utilization U_i of each task τ_i can be expressed as:

$$U_i = \max(U_i^{\max} - \lambda E_i, U_i^{\min}) \tag{4.5}$$

Then since $T_i = C_i/U_i$, we define

$$T_i^{\min} = C_i/U_i^{\max} \tag{4.6}$$

and

$$T_i^{\max} = C_i/U_i^{\min} \tag{4.7}$$

It follows that for $\lambda < C_i/(E_i T_i^{\min})$:

$$T_i = \min\left(\frac{C_i T_i^{\min}}{C_i - \lambda E_i T_i^{\min}}, T_i^{\max}\right) \tag{4.8}$$

Alternatively, given $\lambda_{\max}$ as defined in Equation 3.2, we can express T_i as follows:

$$T_i = \begin{cases} \frac{C_i T_i^{\min}}{C_i - \lambda E_i T_i^{\min}} & \text{if } 0 \leq \lambda < \lambda_i^{\max} \\ T_i^{\min} & \text{if } \lambda \geq \lambda_i^{\max} \end{cases} \tag{4.9}$$

In an uncompressed state, $\lambda = 0$ and for each task, $U_i = U_i^{\max}$ (equivalently, $T_i = T_i^{\min}$).

Baruah [13] also introduces the notation $\Gamma(\lambda)$, representing the task system obtained from Γ by applying compression λ, i.e., with each task τ_i having a period T_i according to Equation 4.8. An optimal algorithm, then, for elastic scheduling of constrained-deadline task systems under EDF finds the value λ^* representing the minimum value λ for which $\Gamma(\lambda)$ is schedulable. In [13], Baruah presents two algorithms that, while not optimal, are nonetheless tunable by a parameter ϵ; both algorithms are guaranteed to find a value $\lambda < \lambda^* + \epsilon$ for which $\Gamma(\lambda)$ is schedulable. We summarize both:

Elastic

This algorithm iterates over values of $\lambda \in [0, \lambda_{\max}]$ with granularity ϵ. For each value of λ tested, it performs PDA over the task set $\Gamma(\lambda)$. Once PDA indicates schedulability, the search stops, and compression is applied. For efficiency, binary search is proposed as an alternative. For a considered value of λ, if PDA indicates schedulability, a smaller value of λ is subsequently tested; if not, a larger value is checked. The binary search limits the number of times PDA is performed to $\lceil \log_2 \left(\frac{\lambda_{\max}}{\epsilon} \right) \rceil$; PDA is itself pseudo-polynomial for bounded-utilization task systems.

Elastic-Efficient

A more efficient algorithm is supported by two observations in [13], repeated here:

Observation 1. *If a given sporadic task system Γ satisfies Expression 4.4 for a given value of t (say, t_o), then any task system Γ' obtained from Γ by increasing the period parameters of one or more tasks also satisfies Expression 4.4 for t_o.*

Observation 2. *Let Γ denote some constrained-deadline elastic sporadic task system, and λ, ϵ, and t_s denote positive numbers. If all elements in the testing set of $\Gamma(\lambda)$ that are $\leq t_s$ satisfy Condition 4.4, then all elements in the testing set of $\Gamma(\lambda+\epsilon)$ that are $\leq t_s$ also satisfy Condition 4.4.*

The algorithm proceeds by iterating over values of λ, beginning with $\lambda = 0$. It considers elements of the PDA testing set in increasing order. Baruah observes that the testing set need not be enumerated in its entirety a priori [13]; instead, the current element being tested, t_o, can be updated to the smallest value from amongst the *next* deadlines of each task:

$$t_o \leftarrow \min_i \left(\left\lfloor \frac{t_o - D_i}{T_i} \right\rfloor + 1 \right) \times T_i + D_i \tag{4.10}$$

When an element is reached for which PDA fails at the current test set element t_o, λ is incremented by ϵ. Observation 2 implies that once PDA succeeds at t_o, only larger values — the next one obtained by Equation 4.10 for the periods T_i of task set $\Gamma(\lambda)$ — need to be tested. Once the test set is exhausted, the current value λ is returned. However, if λ reaches $\lambda_{\max}$, the algorithm terminates, as the task system remains unschedulable even under compression.

Because the algorithm essentially performs a single PDA (the testing set is only traversed once), while additionally recomputing the periods of each task τ_i in $\Gamma(\lambda)$ for each value of λ, the worst-case running time of the algorithm is:

$$O\left(n \times \left\lceil \frac{\lambda_{\max}}{\epsilon} \right\rceil\right) + \text{ the running time of PDA.}$$

where n denotes the number of tasks in Γ. For constant ϵ, this is dominated by the running time of PDA.

4.3 Extension to Fixed-Priority Scheduling

In this section, we present a simple extension of Baruah's algorithm Elastic [13, Algorithm 1] (summarized in Section 4.2.2) to fixed-priority deadline-monotonic (DM) scheduling, which maintains the priority order of tasks as their periods are extended. The procedure is outlined in Algorithm 4. Given a system Γ of elastic constrained-deadline tasks (characterized as described in Section 4.2), it seeks to determine the smallest value of λ for which $\Gamma(\lambda)$ is schedulable. Like the Elastic algorithm, its precision is tunable by a parameter ϵ; the value λ found is guaranteed to be less than $\lambda^*+\epsilon$.

The algorithm initializes λ, the amount of compression to be applied, to 0. It then increases λ in steps of size ϵ, performing RTA for the complete task set $\Gamma(\lambda)$ for each value of λ. Once λ is found for which schedulability is achieved, the algorithm terminates and the value is returned. However, if λ exceeds $\lambda_{\max}$, the utilization constraints on each task prevent the system from being scheduled under the elastic model.

Algorithm 4: Elastic-FP(Γ)

Input: Elastic constrained-deadline task system Γ
Output: Smallest λ such that $\Gamma(\lambda)$ is DM-schedulable
$\lambda \leftarrow 0$
$\lambda_{\max}$ computed according to Equation 3.2
repeat
  Perform RTA for $\Gamma(\lambda)$
  if $\Gamma(\lambda)$ *is schedulable* **then**
    return λ
  else
    $\lambda \leftarrow \lambda + \epsilon$
until $\lambda > \lambda_{\max}$;
return FAILURE

4.3.1 Running Time

Since at most $\lceil \lambda_{\max}/\epsilon \rceil$ calls are made to RTA by the algorithm ELASTIC-FP, its worst-case running time is $\lceil \frac{\lambda_{\max}}{\epsilon} \rceil \times$ the worst-case running time of RTA (which is pseudo-polynomial in the representation of the task system for bounded utilizations).

4.4 An Efficient Iterative Approach

In this section, we present the first of two refinements to Algorithm 4 (ELASTIC-FP). We begin with a brief summary of Audsley et al.'s response time analysis (RTA) [6], which will provide a key observation leveraged by both refinements.

4.4.1 Response-Time Analysis

A task set Γ is schedulable if and only if the response time of each task does not exceed its deadline. Under fixed-priority preemptive scheduling, the response time R_i of a task τ_i is characterized as the sum of its execution time C_i and the interference I_i of the higher priority tasks; the interference is, itself, a function of the response time. Assuming without loss of generality that tasks are indexed by decreasing priority, the following expression describes the response time:

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{4.11}$$

Audsley et al. [6] describe a recursive process by which to determine the response time; the system is schedulable if and only if R_i does not exceed the deadline D_i for each task τ_i. If used in the context of RTA, our algorithm requires up to $\lceil \lambda_{\max}/\epsilon \rceil$ calls to RTA in the worst case. However, the following observation provides a slight improvement to execution time:

Observation 3. *If the condition $R_i \leq D_i$ holds for a task τ_i in $\Gamma(\lambda)$, then the condition also holds for the same task τ_i in $\Gamma(\lambda + \delta)$ for any $\delta > 0$.*

Proof. Since the period T_i only appears in the denominator in the expression for computing response time (Equation 4.11), and the period does not decrease as λ increases (from Equation 4.9), it follows that R_i does not increase when increasing λ. Therefore, if $R_i \leq D_i$ for some λ, the inequality still holds as λ increases. $\square$

4.4.2 The Algorithm

This observation implies that Algorithm 4 can be improved by considering only a *single* task at a time. RTA requires checking every task in a task system, but once a task is shown to be schedulable for a given value of λ, it need not be rechecked for larger values. The resulting improvement is outlined in Algorithm 5.

As in Algorithm 4 (Elastic-FP), Elastic-FP-Efficient begins by initializing λ to 0. It considers tasks in turn, beginning with τ_1, the highest-priority task. The algorithm introduces the notation $\tau_i(\lambda)$ to refer to task $\tau_i \in \Gamma(\lambda)$, i.e., task τ_i having a period T_i according to Equation 4.9 for the given value of λ. When RTA determines that the current task under consideration, τ_i, is unschedulable for the current value of λ, the algorithm increases λ in steps of ϵ until the task is schedulable. At this point, it considers the next task in the system. If there are no tasks remaining to be checked (line 9), the algorithm terminates and returns the current value of λ. However, if the value of λ exceeds $\lambda^{\max}$, the algorithm fails.

Algorithm 5: Elastic-FP-Efficient(Γ)

Input: Elastic constrained-deadline task system Γ
Output: Smallest λ such that $\Gamma(\lambda)$ is DM-schedulable
$\lambda \leftarrow 0$
$\lambda_{\max}$ computed according to Equation 3.2
$i \leftarrow 1$
repeat
    Perform RTA for task $\tau_i(\lambda)$
    if $\tau_i(\lambda)$ *is schedulable* **then**
        if $i == n$ **then**
            return λ
        $i \leftarrow i + 1$
    else
        $\lambda \leftarrow \lambda + \epsilon$
until $\lambda > \lambda_{\max}$;
return Failure

4.4.3 Running Time

As before, at most $\lceil \lambda_{\max}/\epsilon \rceil$ values of λ are checked by the algorithm Elastic-FP-Efficient. However, RTA is only performed for a *single task* at a time. For a single value of λ, no more than a single failing check can be made. Additionally, each task need only have a single successful check. Therefore, for a task system Γ of size n, the total running time of the algorithm can be expressed as:

$$\left(\left\lceil \frac{\lambda_{\max}}{\epsilon} \right\rceil + n - 1 \right) \times \text{ the running time of RTA for a single task} \tag{4.12}$$

4.5 A Binary Search Implementation

Our second refinement to Algorithm 4 (Elastic-FP) instead performs binary search over values of λ in $[0, \lambda_{\max}]$. Observation 3 implies that, when testing using RTA to test $\Gamma(\lambda)$ for schedulability, any tasks already known to be schedulable for smaller values of λ do not need to be rechecked. The complete procedure is outlined in Algorithm 6.

Algorithm 6: Elastic-FP-BS(Γ)

Input: Elastic constrained-deadline task system Γ
Output: Smallest λ such that $\Gamma(\lambda)$ is DM-schedulable
$\lambda_{\text{LO}} \leftarrow 0$
$\lambda_{\max}$ computed according to Equation 3.2
S ▷ `Set of schedulable task indices`
Determine $\Gamma(\lambda_{\max})$ ▷ `Using Equation 4.9`
Perform RTA for $\Gamma(\lambda_{\max})$
if $\Gamma(\lambda_{\max})$ *is schedulable* **then**
    $\lambda_{\text{HI}} \leftarrow \lambda_{\max}$
else
    return FAILURE
repeat
    $\lambda \leftarrow (\lambda_{\text{HI}} + \lambda_{\text{LO}})/2$
    SCHEDULABLE $\leftarrow$ TRUE
    $S' \leftarrow S$
    forall $\tau_i \in \Gamma, i \notin S$ **do**
        Perform RTA for $\tau_i(\lambda)$
        if $\tau_i(\lambda)$ *is schedulable* **then**
            Add i to S'
        else
            SCHEDULABLE $\leftarrow$ FALSE
    if SCHEDULABLE **then**
        $\lambda_{\text{HI}} \leftarrow \lambda$
    else
        $\lambda_{\text{LO}} \leftarrow \lambda$
        $S \leftarrow S'$
until $\lambda_{\text{HI}} - \lambda_{\text{LO}} \leq \epsilon$;
return λ_{HI}

The algorithm first performs RTA for $\Gamma(\lambda_{\max})$; if it is not schedulable, the algorithm fails. Otherwise, it performs binary search over values of λ in the range $[0, \lambda_{\max}]$: λ_{HI} (initialized to $\lambda_{\max}$) tracks the smallest value of λ tested for which $\Gamma(\lambda)$ is schedulable, while λ_{LO} (initialized to 0) tracks the largest tested value for which $\Gamma(\lambda)$ is *not* schedulable. A variable S tracks the indices of tasks in $\Gamma(\lambda_{\text{LO}})$ that *are* schedulable.[10] At each step, the algorithm performs RTA for those tasks $\tau_i \in \Gamma(\lambda)$ that are *not* in S. If all tasks are schedulable, λ_{HI} is decreased to the tested value of λ; otherwise, λ_{LO} is increased to the tested value of λ and S is updated to include those tasks for which RTA nonetheless succeeded. The algorithm terminates when the difference between λ_{HI} and λ_{LO} does not exceed ϵ, at which point it is guaranteed that $\lambda_{\text{HI}} < \lambda^* + \epsilon$ for optimal λ^*, since $\lambda^* > \lambda_{\text{LO}}$.

4.5.1 Running Time

Since algorithm Elastic-FP-BS requires RTA to be performed for all tasks in $\Gamma(\lambda_{\max})$ prior to the binary search, in the worst case $\lceil \log_2(\lambda_{\max}/\epsilon) \rceil + 1$ total calls are made to RTA. The use of the variable S to track tasks already known to be schedulable for smaller values of λ may improve the execution time for some task sets; indeed, if $\lambda^* > \lambda_{\max} - \epsilon$, and if RTA determines schedulability for all but one task at $\lambda = \lambda_{\max}/2$, then binary search will only proceed upward, and RTA will only need to be performed for a single task at each checked value of λ thereafter. In this case, RTA for a *single* task is performed only

$$\left\lceil \log_2 \left(\frac{\lambda_{\max}}{\epsilon} \right) \right\rceil + 2n - 1$$

times. This is more efficient than Algorithm 5 (Elastic-FP-Efficient) if:

$$\left\lceil \log_2 \left(\frac{\lambda_{\max}}{\epsilon} \right) \right\rceil + n < \left\lceil \frac{\lambda_{\max}}{\epsilon} \right\rceil \tag{4.13}$$

For ϵ chosen such that $\lambda_{\max}/\epsilon = 1000$, this is more efficient for systems of fewer than 990 tasks.

On the other hand, if $\lambda^* < \epsilon$, binary search will only proceed downward, so S remains empty, and so RTA must be performed for *all* tasks in Γ for each tested value of λ. In this case,

[10] S can be implemented as a bitmask or an array, allowing $\mathcal{O}(1)$ checking and insertion for a given task index.

RTA for a single task is performed

$$\left(\left\lceil \log_2\left(\frac{\lambda_{\max}}{\epsilon}\right)\right\rceil + 1\right) \times n$$

times. This is more efficient than Algorithm 5 (Elastic-FP-Efficient) only if:

$$\left\lceil \log_2\left(\frac{\lambda_{\max}}{\epsilon}\right)\right\rceil \times n + 1 < \left\lceil \frac{\lambda_{\max}}{\epsilon} \right\rceil \tag{4.14}$$

For ϵ chosen such that $\lambda_{\max}/\epsilon = 1000$, this is guaranteed to be more efficient only if the system has fewer than 100 tasks. In Section 4.8, we evaluate both algorithms to compare their performance in the context of randomly-generated synthetic task sets.

4.6 An MIQP Representation

In this section we describe how to formulate the problem of finding the value λ^* representing the minimum necessary compression to achieve schedulability for a fixed-priority, preemptive task system Γ as a mixed integer quadratic program (MIQP).

4.6.1 Formulating the MIQP

We build upon the mixed integer linear programming representation of RTA given in [17]. In [80], it is shown that for a fixed-priority, preemptive task system Γ to be schedulable, it is necessary and sufficient that for each $\tau_i \in \Gamma$, there exists some value of $t \leq D_i$ for which:[11]

$$t \geq C_i + \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil \times C_j \tag{4.15}$$

The minimum value of t satisfying this condition for τ_i is the *response time* of the task. However, unlike in [17], we do not seek to find the minimum value of t for each task. Instead, we intend to find the minimum value of λ for which there exists a $t_i < D_i$ for each task τ_i

[11] Without loss of generality, we again assume tasks are indexed in decreasing order of priority.

satisfying Expression 4.15. In other words, it must satisfy:

$$t_i \geq C_i + \sum_{j=1}^{i-1} \left\lceil \frac{t_i}{T_j(\lambda)} \right\rceil \times C_j \tag{4.16}$$

for $T_j(\lambda)$ as defined in Equation 4.9. The MIQP problem is formulated as follows:

1. We define a real-valued variable λ representing the compression applied to all tasks τ_i in the task system Γ.

2. To enforce this intended interpretation, we specify the constraint:

$$0 \leq \lambda \leq \lambda^{\max} \tag{4.17}$$

   where $\lambda^{\max}$ is as defined in Equation 3.2. We note that, because we are minimizing λ, the constraint on its upper bound is not strictly necessary: a value $\lambda' > \lambda^{\max}$ will not achieve greater compression than $\lambda^{\max}$, and so the smaller $\lambda^{\max}$ would be selected instead of λ' anyway. However, we observe that this constraint tightens the search space, slightly improving the solver's execution time.

3. For each task τ_i, we define a real-valued variable t_i representing some value of t for which Expression 4.15 holds. Unlike in [17], where the corresponding R_i is non-negative, we restrict t_i to be positive: if $t_i = 0$ satisfies Expression 4.15, then $C_i = 0$, in which case the task can be ignored. This becomes an important distinction in a later step.

4. We therefore specify the constraint:

$$0 \leq t_i \leq D_i \tag{4.18}$$

5. As in [17], for each task τ_i and every $j \in \{1, \ldots, i - i\}$, we define a non-negative *integer* variable Z_{ij} that represents the term $\lceil t_i / T_j(\lambda) \rceil$.

6. As in [17], to enforce this intended interpretation on the Z_{ij} variables, for every task τ_i we must add constraints of the form $Z_{ij} \geq t_i / T_j(\lambda)$ for each $j \in \{1, \ldots, i - i\}$. Since Z_{ij} is specified to be an integer variable, this will respect the ceiling operator that appears in Expression 4.15.

From the formulation of $T_j(\lambda)$ in Equation 4.9, we can express the term $\lceil t_i/T_j(\lambda)\rceil$ as:

$$\max\left(\left\lceil \frac{t_i}{T_j^{\max}} \right\rceil, \left\lceil \frac{t_i(C_j - \lambda E_j T_j^{\min})}{C_j T_j^{\min}} \right\rceil\right)$$

This requires two constraints for each Z_{ij}:

$$\forall j \in \{1, \ldots, i-1\},\ Z_{ij} \geq \left(\frac{t_i}{T_j^{\max}}\right) \tag{4.19}$$

$$\forall j \in \{1, \ldots, i-1\},\ Z_{ij} \geq \left(\frac{t_i(C_j - \lambda E_j T_j^{\min})}{C_j T_j^{\min}}\right)$$

Because t_i is itself a variable, and λ is the value we ultimately want to minimize, we rewrite the second expression as:

$$0 \leq Z_{ij} - \left(\frac{1}{T_j^{\min}}\right) t_i + \left(\frac{E_j}{C_j}\right) t_i \times \lambda \tag{4.20}$$

This is a quadratic constraint, as it contains the term $t_i \times \lambda$.

7. As in [17], for each task τ_i we add a final constraint representing Expression 4.16:

$$C_i + \sum_{j=1}^{i-1} Z_{ij} \times C_j \leq t_i \tag{4.21}$$

8. To find the minimum value of λ for which the problem can be satisfied, we add the following **objective function**:

$$\textbf{minimize } \lambda \tag{4.22}$$

Recall that in Step 1 of the MIQP, t_i is restricted to the positive reals; this implies that if a solution exists, λ is well-defined for any value of t_i.

4.6.2 The Resulting Algorithm

For completeness, we present an algorithm to apply this MIQP to find the optimal value, λ^*, representing the minimum amount of compression to guarantee schedulability of a set of tasks Γ. The procedure is outlined in Algorithm 7.

Algorithm 7: Elastic-FP-MIQP-Joint(Γ)

```
Input: Elastic constrained-deadline task system Γ
Output: Smallest λ such that Γ(λ) is DM-schedulable
forall τ_i ∈ Γ(0) do
    Perform RTA for τ_i
    if τ_i is not schedulable then
        Construct MIQP
        Solve for λ*
        if MIQP infeasible then return FAILURE
        else return λ*
```

4.6.3 Problem Size and Running Time

The MIQP described in this section to find λ^* for the complete set Γ of tasks τ_i has a single real variable λ and real variables t_i for each task τ_i. λ has a linear constraint, and each t_i has two linear constraints. The problem also has an integer variable Z_{ij} for every task τ_i and every value of $j < i$; each such variable has a single linear constraint and a single quadratic constraint. The problem therefore has $n + 1$ real variables and $\binom{n}{2} = (n^2 - n)/2$ integer variables overall, with $2n + \binom{n}{2} = (n^2 + 3n)/2$ total linear constraints and $(n^2 - n)/2$ total quadratic constraints. Table 4.1 summarizes the problem size.

Real Variables	$n + 1$
Integer Variables	$\frac{n^2-n}{2}$
Linear Constraints	$\frac{n^2+3n}{2}$
Quadratic Constraints	$\frac{n^2-n}{2}$
Times to Run	1

Table 4.1: Number of variables and constraints for the MIQP.

Notice that the number of quadratic constraints itself grows quadratically with the number of tasks. As we demonstrate in Section 4.8.3, this makes the problem difficult to solve using our choice of off-the-shelf solver, SCIP [2]. We note that even the decision version of this problem (showing the existence of a λ that satisfies schedulability) is NP-complete, since verifying a given value of λ requires performing RTA for the resulting task system $\Gamma(\lambda)$. In the next section, we present a more efficient (though still NP-hard) MIQP-based approach.

4.7 Simplifying the Problem: An MIQP Per Task

This section modifies the MIQP of the previous section to find just the value λ_i^* representing the minimum compression necessary to guarantee schedulability of a *single* task τ_i. We then use this result to present an algorithm that finds the optimal compression value λ^* for the complete task system.

4.7.1 Formulating the MIQP

The MIQP to find the value λ_i^* to guarantee schedulability of *just* the task τ_i in a set Γ is formulated as follows:

1. We introduce a variable λ_i representing the compression applied to the task system Γ.
2. To enforce this intended interpretation, we again specify the constraint:
$$0 \leq \lambda \leq \lambda^{\max} \tag{4.23}$$
where $\lambda^{\max}$ is as defined in Equation 3.2. We do *not* constrain λ_i to remain less than $\lambda_i^{\max}$ (notice the index i). For higher priority tasks τ_j, $j < i$, compression $\lambda > \lambda_i^{\max}$ might be necessary to reduce their interference on τ_i sufficiently for τ_i to become schedulable.
3. For the *single* task τ_i, we define a positive real-valued variable t_i representing some value for t for which Expression 4.15 holds.

4. Again, t_i is constrained as:

$$0 \leq t_i \leq D_i \tag{4.24}$$

As before, unless $C_i = 0$, this effectively constrains t_i as $0 < t_i \leq D_i$.

5. As in [17], for each $j \in \{1, \ldots, i - i\}$, we define a non-negative *integer* variable Z_{ij} that represents the term $\lceil t_i / T_j(\lambda_i) \rceil$.

6. As before, to enforce this intended interpretation on the Z_{ij} variables, we must add the following constraints for each Z_{ij}:

$$\forall j \in \{1, \ldots, i-1\},\ Z_{ij} \geq \left(\frac{t_i}{T_j^{\max}} \right) \tag{4.25}$$

$$0 \leq Z_{ij} - \left(\frac{1}{T_j^{\min}} \right) t_i + \left(\frac{E_j}{C_j} \right) t_i \times \lambda_i \tag{4.26}$$

The latter is a quadratic constraint, as it contains the term $t_i \times \lambda_i$.

7. As before, we add a final constraint to enforce the relationship in Expression 4.16:

$$C_i + \sum_{j=1}^{i-1} Z_{ij} \times C_j \leq t_i \tag{4.27}$$

8. To find the minimum value of λ_i for which the problem can be satisfied, we add the following **objective function**:

$$\textbf{minimize } \lambda_i \tag{4.28}$$

4.7.2 The Resulting Algorithm

By solving an MIQP, we can find the exact value of λ_i by which to compress a task system Γ to guarantee schedulability for an *individual* task according to RTA. We now present an algorithm that extends this approach to finding the optimal value, λ^*, representing the minimum amount of compression to guarantee schedulability of *all* tasks. The procedure is outlined in Algorithm 8.

Algorithm 8: Elastic-FP-MIQP(Γ)

Input: Elastic constrained-deadline task system Γ
Output: Smallest λ such that $\Gamma(\lambda)$ is DM-schedulable

$\lambda \leftarrow 0$
forall $\tau_i \in \Gamma$ **do**
    Perform RTA for $\tau_i(\lambda)$
    if $\tau_i(\lambda)$ *is* ***not*** *schedulable* **then**
        Construct MIQP
        Solve for λ_i^*
        if *MIQP infeasible* **then**
            return FAILURE
        $\lambda \leftarrow \lambda_i^*$
return λ

The algorithm initializes λ to 0. Each task in the system is checked for schedulability under the current compression level using RTA. Once a task τ_i is found that cannot be scheduled, the MIQP is constructed (which requires calculating $\lambda_i^{\max}$) and solved for that task. If no solution is found, the minimum utilization constraints of its constituent tasks prevent the system from compressing to schedulability. Otherwise, λ is updated to λ_i^*, and the remaining tasks are checked in the same manner.

4.7.3 Problem Size

The MIQP described this section for a single task τ_i has real variables λ_i and t_i, as well as an integer variable Z_{ij} for every $j < i$. The variable t_i has two linear constraints to enforce it remains less than the deadline D_i and to enforce the recurrence in Expression 4.15. To enforce the ceiling function in Expression 4.15, and to represent the relationship between the task period T_i and compression λ_i, each variable Z_{ij} has a single linear constraint and a single quadratic constraint. Thus, for a set Γ of n tasks, this MIQP has up to 2 real variables, up to $n-1$ integer variables, up to $n+1$ linear constraints, and up to $n-1$ quadratic constraints. This is linear in the number of tasks; in comparison, the MIQP described in Section 4.6.1 has constraints quadratic in the number of tasks. However, compared to the single MIQP, the problem presented in this section has to be solved up to n times. Table 4.2 summarizes this comparison.

	MIQP per Task	Single MIQP
Real Variables	2	$n+1$
Integer Variables	$\leq n-1$	$\frac{n^2-n}{2}$
Linear Constraints	$\leq n+1$	$\frac{n^2+3n}{2}$
Quadratic Constraints	$\leq n-1$	$\frac{n^2-n}{2}$
Times to Run	$\leq n$	1

Table 4.2: A comparison of compressing with an MIQP per task versus a single MIQP for the entire set of tasks.

Nonetheless, this version of the problem should be easier to solve than the single MIQP, as the execution time to solve a quadratic program typically scales superlinearly with the number of constraints. This is backed by the results of our evaluations in Section 4.8.3.

4.8 Evaluation

To evaluate the effectiveness and efficiency of the Elastic-FP-Efficient (Algorithm 5), Elastic-FP-BS (Algorithm 6), Elastic-FP-MIQP-Joint (Algorithm 7), and Elastic-FP-MIQP (Algorithm 8) procedures, we run them over a large collection of randomly-generated constrained-deadline task sets. We track their execution, both in time and calls to RTA. We also analyze the overhead incurred by using these algorithms for online admission control in an embedded system — the same Raspberry Pi 3 Model B+ used in Section 2.4.1. Finally, we compare the compression values λ produced by each algorithm.

4.8.1 Generating Task Sets

To evaluate elastic scheduling for constrained-deadline tasks, we consider tasks that *begin* with a relative deadline D_i equal to their period T_i. Tasks individually have a utilization not exceeding 1, but the task systems *as a whole* are not schedulable due to their joint utilizations exceeding the utilization bound of the system. To accommodate such a task system, each task has its *period* extended while its *deadline* remains the same. We generate task sets of size [10−100] in steps of 10. For each size, we consider total utilizations in the range [1.0−2.0] in steps of 0.1. For each combination of size/utilization we generate 100 sets of tasks according to the following methodology:

1. Uncompressed task periods T_i^{min} are generated using a log-uniform distribution (per [61]) in [1−1000].

2. Task deadlines D_i are set equal to T_i^{min}.

3. Tasks are sorted according to increasing deadline (decreasing priority under DM scheduling).

4. The total utilization of the task system is distributed in an unbiased random fashion among tasks using the Dirichlet Rescale (DRS) algorithm [70], such that each task is assigned a utilization U_i^{max}.

5. Execution time is assigned according to $C_i{=}U_i^{\text{max}}{\cdot}T_i^{\text{min}}$.

6. Elasticity E_i is uniformly selected in [0−1].

For each task set thus generated, a minimum utilization U_i^{min} is assigned to each task so that the *total* minimum utilization cannot exceed 0.69, the Liu and Layland schedulability bound of a rate-monotonic implicit-deadline task system [97]. We do so in two different ways:

1. Scale: For half of the tasks, we define a constant $s = 0.69/U_{\text{SUM}}^{\text{max}}$, where $U_{\text{SUM}}^{\text{max}}$ is the maximum total utilization of the task set. We then obtain U_i^{min} by multiplying each task's U_i^{max} by a random value uniformly selected from the range $[0{-}s]$. On average, the total minimum utilization is expected to be 0.345, with a narrower distribution for larger task sets. This is illustrated in Figure 4.1.

2. DRS: For the other half, we apply DRS to distribute the total minimum utilization $U_{\text{SUM}}^{\text{min}} = 0.69$ across the individual U_i^{min} values uniformly from the space of selections satisfying the conditions that *(i)* the total $\sum_i U_i^{\text{min}} = 0.69$ and *(ii)* each value U_i^{min} does not exceed the corresponding U_i^{max}.

Each task's maximum period is then derived as $T_i^{\text{max}} = U_i^{\text{min}}/C_i$.

The result is that, for every combination of size/utilization, we now have 200 sets of tasks, for a total of 22 000 overall.

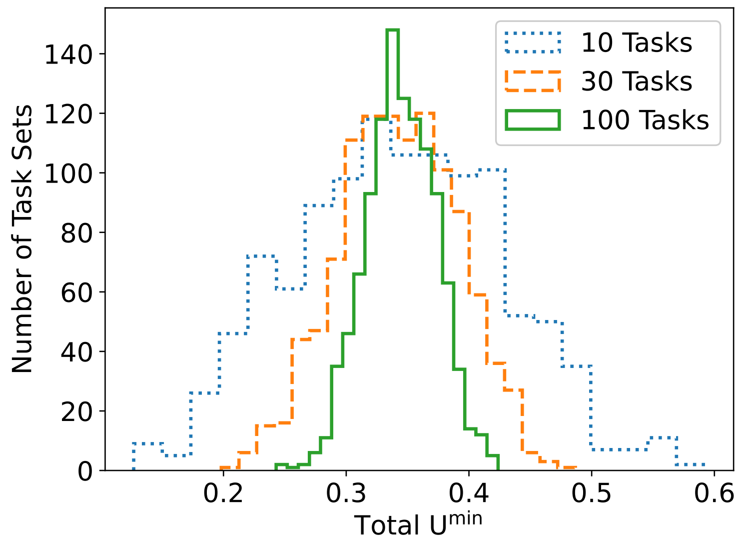


Figure 4.1: Distributions of total minimum utilizations from 1100 randomly-generated task sets each of size 10, 30, 100 with $U_i^{\min}$ values assigned according to method 6(a).

4.8.2 Implementation

We implement our algorithms as written using C++, linking version 8 [22] of the SCIP constraint integer program solver [2] to execute the MIQP. Task parameters C_i, $T_i^{\min}$, $T_i^{\max}$, T_i, D_i, and E_i are stored in global arrays using single-precision floating-point representations. We program the approach of Audsley et al. [6] to perform RTA in an iterative, rather than recursive, style to avoid the time and stack overheads associated with large numbers of nested function calls. We quantify execution time performance by reading from the standard library's high resolution clock (`std::chrono::high_resolution_clock`). Compilation is performed using the Gnu Compiler Collection (GCC) at optimization level `O3`. We enclose calls to the algorithm between calls to `std::atomic_signal_fence` using sequentially-consistent ordering; this avoids instruction reordering around clock reads.

4.8.3 Offline Execution Efficiency

Elastic scheduling can be leveraged in situations where a task set is compressed offline for scheduling on an overutilized target system. To consider this case, we evaluate the execution of our algorithms on a server running Linux kernel version 5.4.0 and having two Intel Xeon

Gold 6130 (Skylake) processors running at 2.1 GHz, and with 32GB of memory. For these experiments, HyperThreading remains enabled to leverage all available parallelism provided by the system — 64 logical cores.

Approximate Algorithms

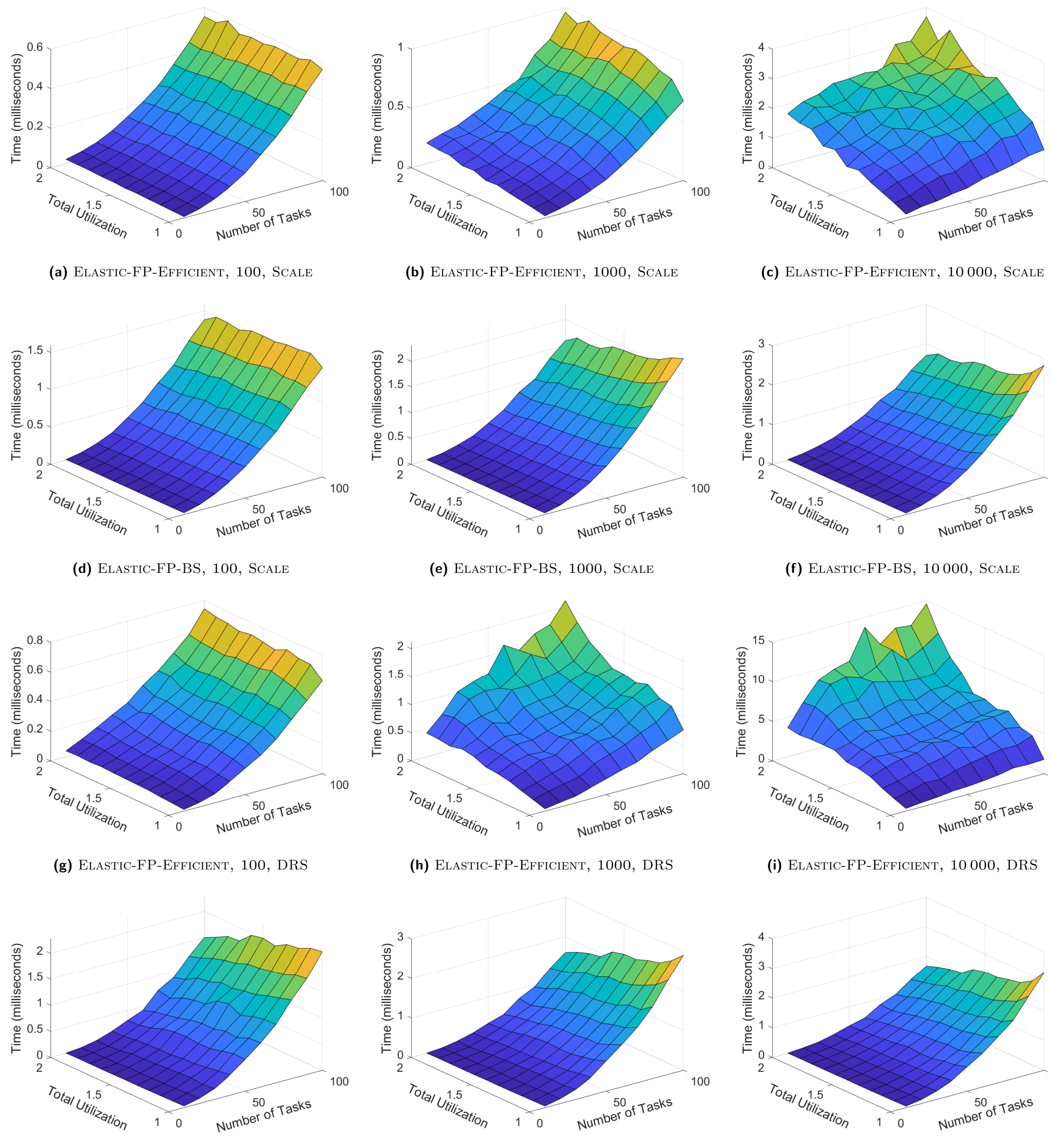
We begin by executing both Elastic-FP-Efficient and Elastic-FP-BS sequentially in a single-threaded environment over all 22 000 task sets. For each task set, we consider values of ϵ that divide the compression limit $\lambda^{\max}$ respectively 100, 1000, and 10 000 times. Mean execution times are illustrated in Figure 4.2, and median times are illustrated in Figure 4.3. ***Both algorithms are quite efficient***, with mean execution times remaining under 4 milliseconds for tasks with minimum utilizations assigned per Scale and under 15 milliseconds for DRS. For larger values of ϵ, iteration is more efficient on average than binary search; but as ϵ gets smaller, Elastic-FP-BS becomes faster. Worst observed execution times (WOET) and maximum total calls to RTA[12] are listed in Table 4.3.

		Scale		DRS	
Algorithm	$\lambda^{\max}/\epsilon$	WOET (ms)	Max RTA Calls	WOET (ms)	Max RTA Calls
Elastic-FP-Efficient	100	0.90	122	1.53	200
Elastic-FP-Efficient	1000	2.14	1021	8.15	1099
Elastic-FP-Efficient	10 000	14.0	10 023	74.1	10 101
Elastic-FP-BS	100	2.38	700	3.50	700
Elastic-FP-BS	1000	3.47	1000	3.95	1000
Elastic-FP-BS	10 000	3.87	1400	5.28	1400

Table 4.3: Algorithm performance comparison on Xeon-based server.

As expected, algorithm running times are closely related to the number of calls to RTA. We also observe that when minimum utilizations are assigned to tasks using the DRS algorithm instead of our Scale method, the worst-observed algorithm execution times are higher, despite the number of calls to RTA remaining approximately the same. This can be explained as follows. While both methods impose the same upper bound (0.69) on the total minimum utilization $U_{\text{SUM}}^{\min}$, on average for Scale, $U_{\text{SUM}}^{\min}$ is half that of DRS. This allows more elastic tasks to compress farther on average, which results in less interference on lower priority tasks, allowing individual calls to RTA to execute more quickly.

[12] Performing response time analysis for a *single* task counts 1 call to RTA.



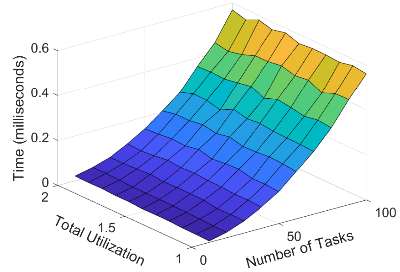
(a) Elastic-FP-Efficient, 100, Scale **(b)** Elastic-FP-Efficient, 1000, Scale **(c)** Elastic-FP-Efficient, 10 000, Scale

(d) Elastic-FP-BS, 100, Scale **(e)** Elastic-FP-BS, 1000, Scale **(f)** Elastic-FP-BS, 10 000, Scale

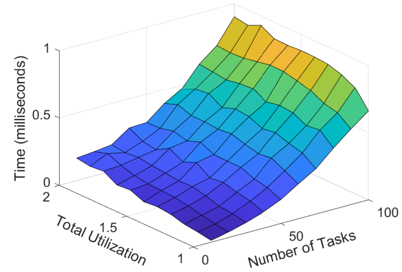
(g) Elastic-FP-Efficient, 100, DRS **(h)** Elastic-FP-Efficient, 1000, DRS **(i)** Elastic-FP-Efficient, 10 000, DRS

(j) Elastic-FP-BS, 100, DRS **(k)** Elastic-FP-BS, 1000, DRS **(l)** Elastic-FP-BS, 10 000, DRS

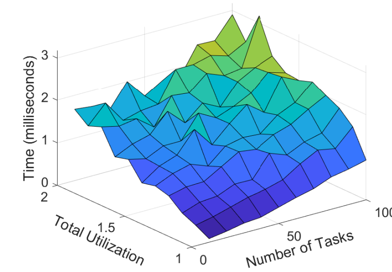
Figure 4.2: Mean algorithm execution times on Intel Xeon Gold 6130.



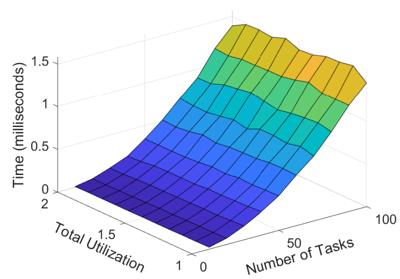
(a) Elastic-FP-Efficient, 100, Scale



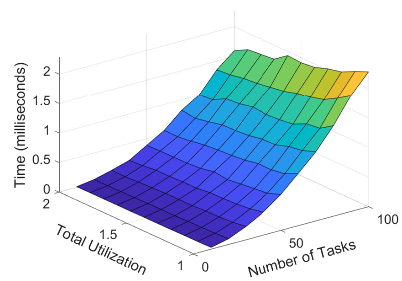
(b) Elastic-FP-Efficient, 1000, Scale



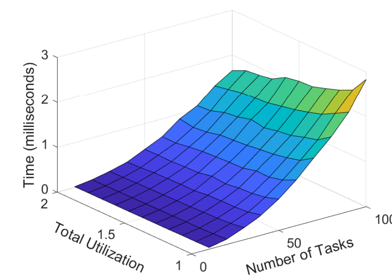
(c) Elastic-FP-Efficient, 10 000, Scale



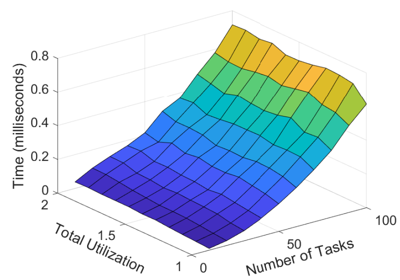
(d) Elastic-FP-BS, 100, Scale



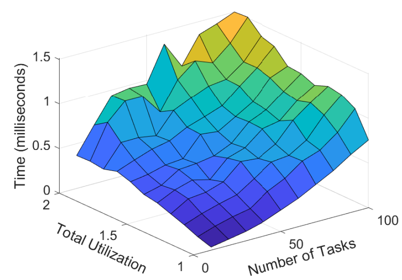
(e) Elastic-FP-BS, 1000, Scale



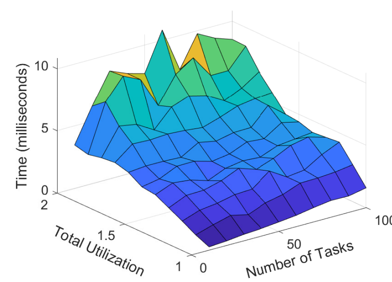
(f) Elastic-FP-BS, 10 000, Scale



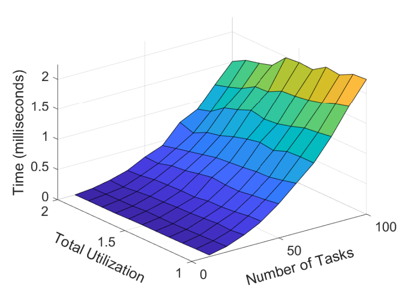
(g) Elastic-FP-Efficient, 100, DRS



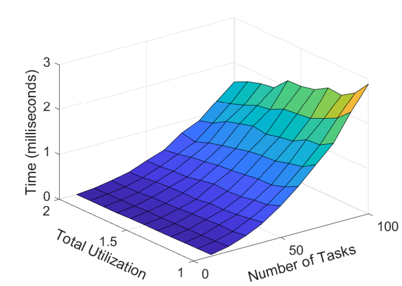
(h) Elastic-FP-Efficient, 1000, DRS



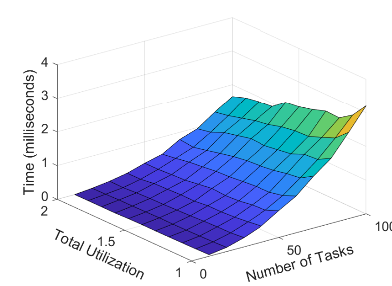
(i) Elastic-FP-Efficient, 10 000, DRS



(j) Elastic-FP-BS, 100, DRS



(k) Elastic-FP-BS, 1000, DRS



(l) Elastic-FP-BS, 10 000, DRS

Figure 4.3: Median algorithm execution times on Intel Xeon Gold 6130.

We also observe from Figures 4.2 and 4.3 that for larger task sets, execution times associated with Elastic-FP-BS *increase* for sets of tasks with lower total maximum utilizations. This is due to the fact that λ^* remains closer to 0, and so the search primarily proceeds downward, meaning that fewer tasks are added to the set S for which RTA can be skipped due to being schedulable at λ_{LO}.

Comparison of MIQP Algorithms

We next compare the performance of Elastic-FP-MIQP-Joint and Elastic-FP-MIQP. We execute both algorithms over just our generated task systems with 10 tasks. The solver is configured to execute in a single thread, which allows us to run separate instances of the algorithm sequentially on 50 of the unused logical cores on our server, splitting up the work of compressing all 2200 considered task systems. The solver is configured to terminate if it has not converged after one hour of execution. Execution time distributions are illustrated in Figure 4.4. Table 4.4 summarizes execution time statistics, including listing the number of times Elastic-FP-MIQP-Joint timed out after an hour.

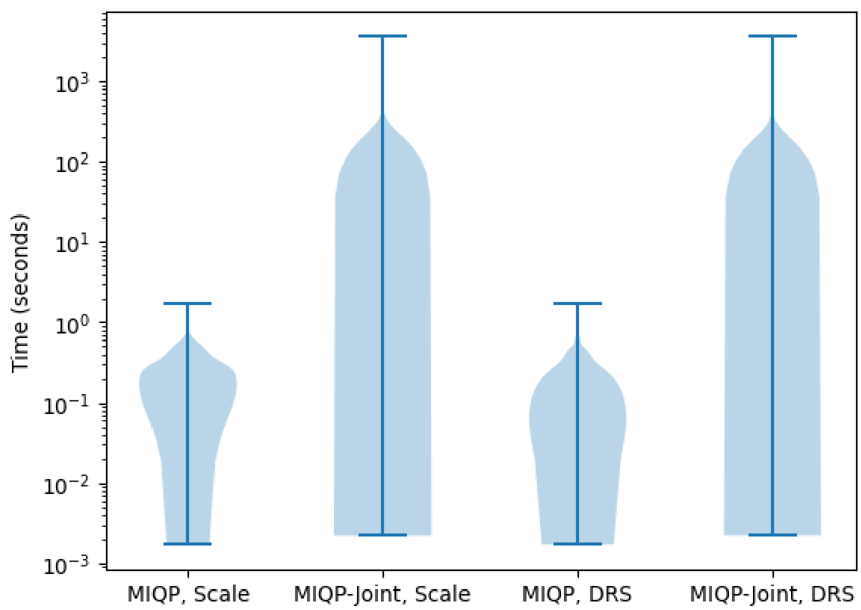


Figure 4.4: Comparison of execution time distributions for Elastic-FP-MIQP-Joint and Elastic-FP-MIQP for 2200 sets of 10 tasks.

	Elastic-FP-MIQP-Joint		Elastic-FP-MIQP	
	Scale	DRS	Scale	DRS
Mean Execution Time	117 s	93 s	254 ms	183 ms
Median Execution Time	403 ms	161 ms	220 ms	147 ms
Maximum Execution Time	> 1 hr	> 1 hr	1.69 s	1.73 s
Number of Timeouts	30	25	0	0

Table 4.4: Comparison of Elastic-FP-MIQP-Joint and Elastic-FP-MIQP for 2200 sets of 10 tasks.

We observe that both algorithms complete somewhat faster on average for tasks with minimum utilizations generated according to DRS than for Scale, but that in the worst case, Elastic-FP-MIQP completes slightly faster for Scale, though these differences are not highly pronounced. More importantly, ***the joint MIQP is extremely inefficient compared to solving an MIQP for each individual task.*** For those task sets we considered, the mean execution time for Elastic-FP-MIQP was 459× faster for Scale and 510× faster for DRS compared to Elastic-FP-MIQP-Joint. In the worst case, Elastic-FP-MIQP was at least 2080× faster; since Elastic-FP-MIQP-Joint timed out for 2.5% of problems, we cannot report the actual speedup.

MIQP Performance for Larger Task Sets

We have observed that the number of constraints used in the quadratic program of Elastic-FP-MIQP-Joint makes it infeasible to use even for offline compression. However, the iterative Elastic-FP-MIQP, which might invoke the quadratic solver for each task, nonetheless uses a simpler expression of the quadratic program that makes it feasible for sets of 10 tasks. To evaluate its scalability to larger task systems, we apply it to our generated task systems of up to 50 tasks. Again, the solver is configured to execute in a single thread, and we split the 11 000 considered task systems over 50 logical cores. Execution time distributions when applied to tasks with minimum utilizations assigned according to Scale are plotted in Figure 4.5, and in Figure 4.6 for DRS.

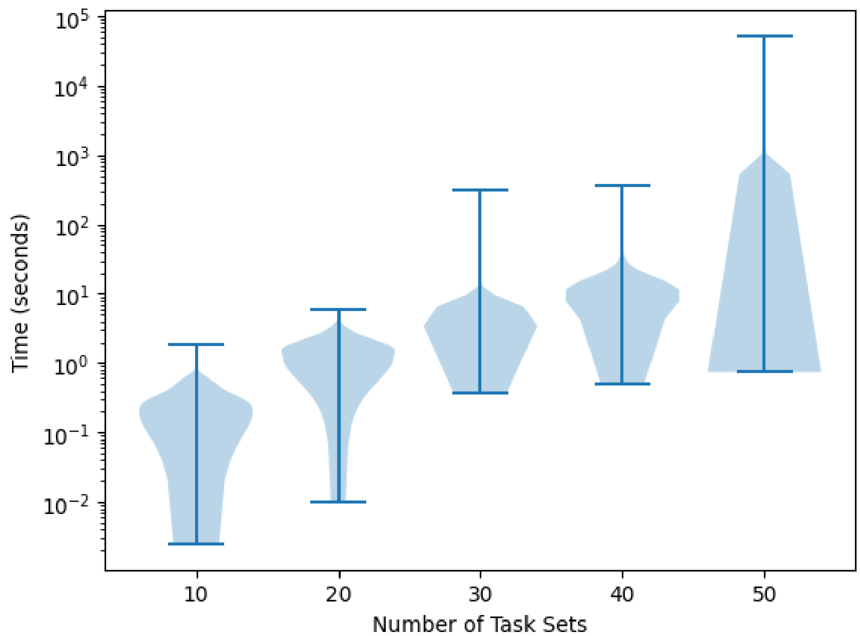


Figure 4.5: Execution time distributions for ELASTIC-FP-MIQP for sets of 10–50 tasks with minimum utilizations assigned per SCALE.

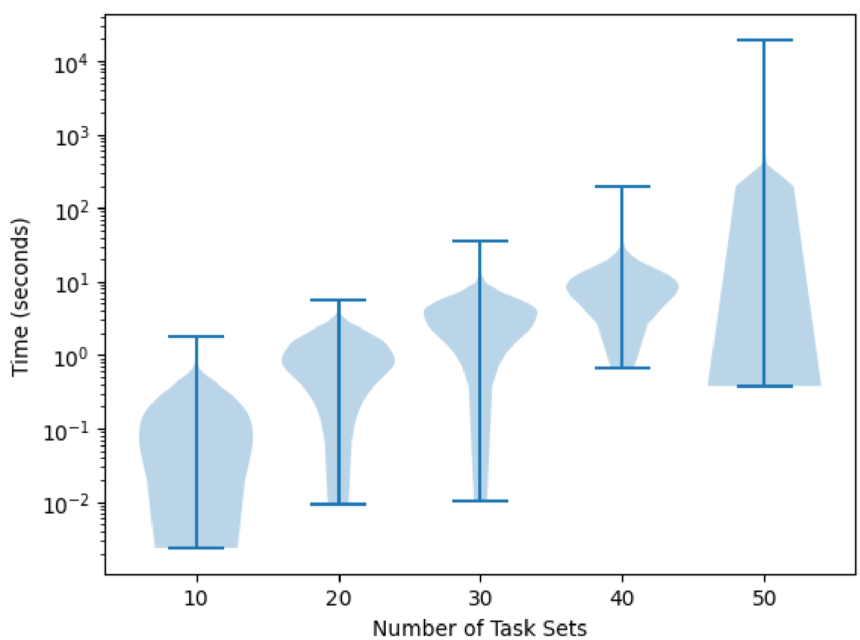


Figure 4.6: Execution time distributions for ELASTIC-FP-MIQP for sets of 10–50 tasks with minimum utilizations assigned per DRS.

For task sets with 20 or fewer tasks, the algorithm consistently completes in under 5.9 seconds. Even with up to 40 tasks, the algorithm finds an optimal value of λ in under 6.1 minutes for SCALE and under 3.3 minutes for DRS, while ***remaining under 26 seconds 95% of the time for sets of 40 tasks.***

However, for systems of 50 tasks, the solver may be slow to produce the optimal solution. For 95% of task sets with 50 tasks, an optimal solution is returned in under 91 seconds for SCALE and under 66 seconds for DRS. However, one of the task sets for SCALE took 14.5 hours to complete, and one for DRS took 5.4 hours. The algorithm took under an hour for the remaining task sets.

4.8.4 Online Execution Efficiency

Elastic scheduling can also be used for online adaption of task rates, e.g., when admitting new tasks on a fully-utilized system, or when task execution times change in response to dynamic exogenous factors. Despite its precision, the uncertain execution times associated with solving the MIQP make the ELASTIC-FP-MIQP algorithm unsuitable for online scheduling decisions in real-time systems. We consider instead the worst observed execution times of the ELASTIC-FP-EFFICIENT and ELASTIC-FP-BS algorithms when running on an embedded system. We apply both algorithms to each of the 22 000 randomly-generated task sets, again using values of $\lambda^{\max}/\epsilon$ in $\{100, 1000, 10\,000\}$.

We measure their execution times on a single core of a Raspberry Pi 3 Model B+, using the same system configuration as the evaluations in Section 2.4.1. Algorithms are compiled statically using version 10.2.1 of the Gnu Compiler Collection (GCC) at optimization level `O3`. To avoid interference from other processes, we disable real-time throttling by writing -1 to the file `/proc/sys/kernel/sched_rt_runtime_us`, isolate CPU core 3 from the scheduler, and run our algorithms on that core at the highest real-time priority under Linux's `SCHED_FIFO` scheduling class.

For each combination of size, maximum utilization, and method of assigning minimum utilizations, we take the worst observed execution time (WOET) among the 100 task sets tested; these are plotted in Figure 4.7, with a summary provided in Table 4.5.

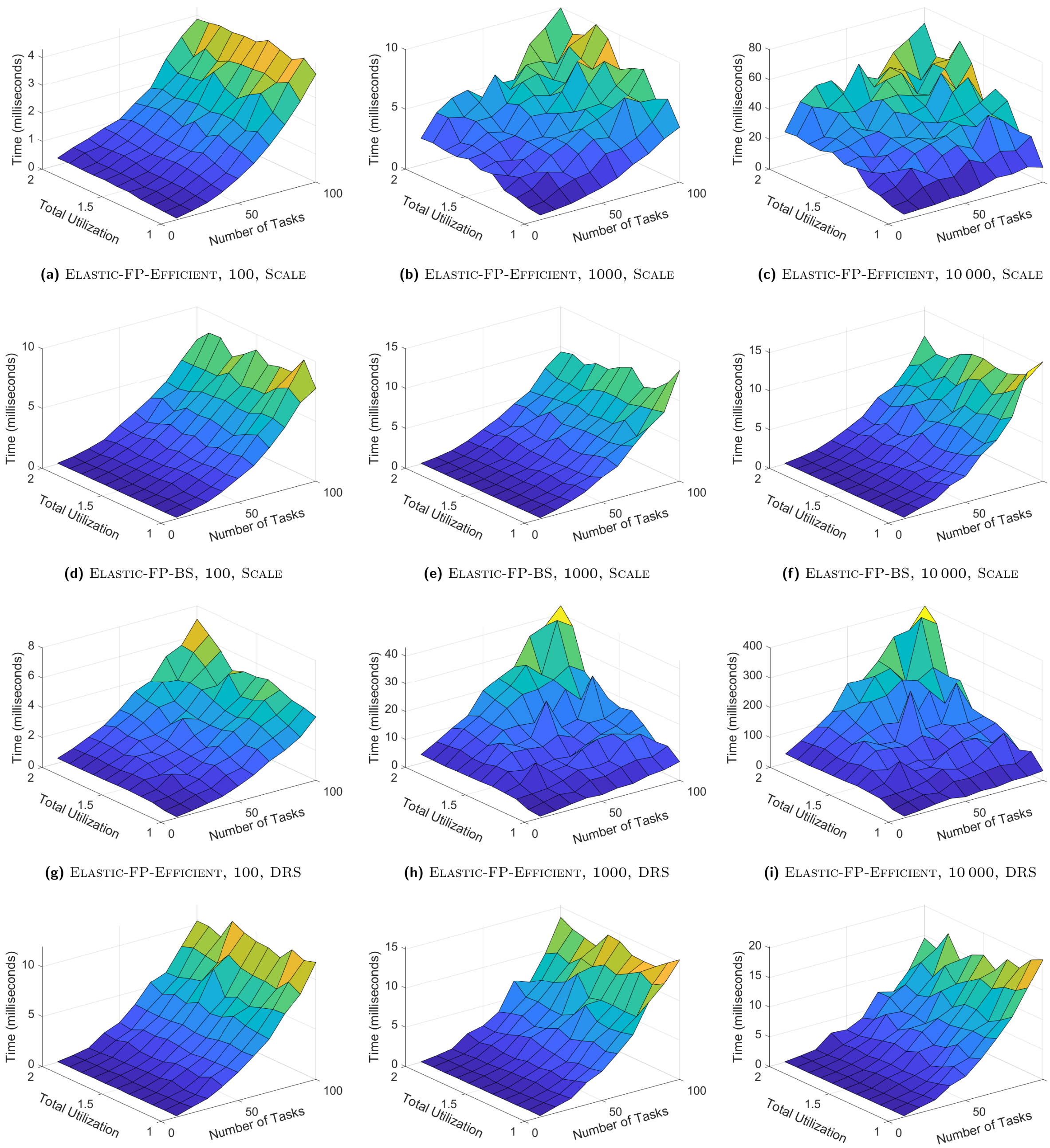
Algorithm	$\lambda^{\max}/\epsilon$	WOET (ms) SCALE	WOET (ms) DRS
ELASTIC-FP-EFFICIENT	100	4.28	7.10
ELASTIC-FP-EFFICIENT	1000	9.98	42.9
ELASTIC-FP-EFFICIENT	10 000	69.6	399
ELASTIC-FP-BS	100	9.69	12.0
ELASTIC-FP-BS	1000	13.9	15.3
ELASTIC-FP-BS	10 000	15.5	20.2

Table 4.5: Algorithm performance comparison on a Raspberry Pi 3B+.

Similar patterns emerge to those we saw in Section 4.8.3. As before, the iterative algorithm outperforms the binary search for larger values of ϵ, but as ϵ decreases, ELASTIC-FP-BS begins to outperform ELASTIC-FP-EFFICIENT. For $\lambda^{\max}/\epsilon = 100$, ELASTIC-FP-EFFICIENT remains under 5 ms even for task sets of size 100 for SCALE and under 8 ms for DRS; and for $\lambda^{\max}/\epsilon = 10\,000$, ELASTIC-FP-BS takes less than 16 ms for SCALE and 21 ms for DRS.

We again observe that the execution times associated with DRS are significantly higher than for SCALE. This is especially true for the iterative ELASTIC-FP-EFFICIENT algorithm with smaller values of ϵ; for $\lambda^{\max}/\epsilon = 10\,000$, the algorithm takes 5.7× longer for task sets with minimum utilizations assigned using DRS compared to the SCALE method.

Nonetheless, ***the low overhead suggests that either algorithm may be effective even for online compression*** of large fixed-priority constrained-deadline task sets on low-power embedded hardware. Appropriate selection — using ELASTIC-FP-EFFICIENT if low granularity is acceptable or ELASTIC-FP-BS if higher granularity is desired — guarantees a solution is found within about 1/50 of a second in the worst case that we observed.



(a) Elastic-FP-Efficient, 100, Scale

(b) Elastic-FP-Efficient, 1000, Scale

(c) Elastic-FP-Efficient, 10 000, Scale

(d) Elastic-FP-BS, 100, Scale

(e) Elastic-FP-BS, 1000, Scale

(f) Elastic-FP-BS, 10 000, Scale

(g) Elastic-FP-Efficient, 100, DRS

(h) Elastic-FP-Efficient, 1000, DRS

(i) Elastic-FP-Efficient, 10 000, DRS

(j) Elastic-FP-BS, 100, DRS

(k) Elastic-FP-BS, 1000, DRS

(l) Elastic-FP-BS, 10 000, DRS

Figure 4.7: Maximum observed execution times on ARM Cortex-A53 (Raspbery Pi 3B+).

4.8.5 Effectiveness of the Approximate Algorithms

When choosing between the four presented algorithms (Elastic-FP-Efficient, Elastic-FP-BS, Elastic-FP-MIQP-Joint, and Elastic-FP-MIQP), one must consider the tradeoffs between *execution time* and *accuracy*. We have already demonstrated that solving Elastic-FP-MIQP-Joint rapidly becomes infeasible as the number of tasks grows; we therefore exclude it from further comparison.

Relative Compression Achieved

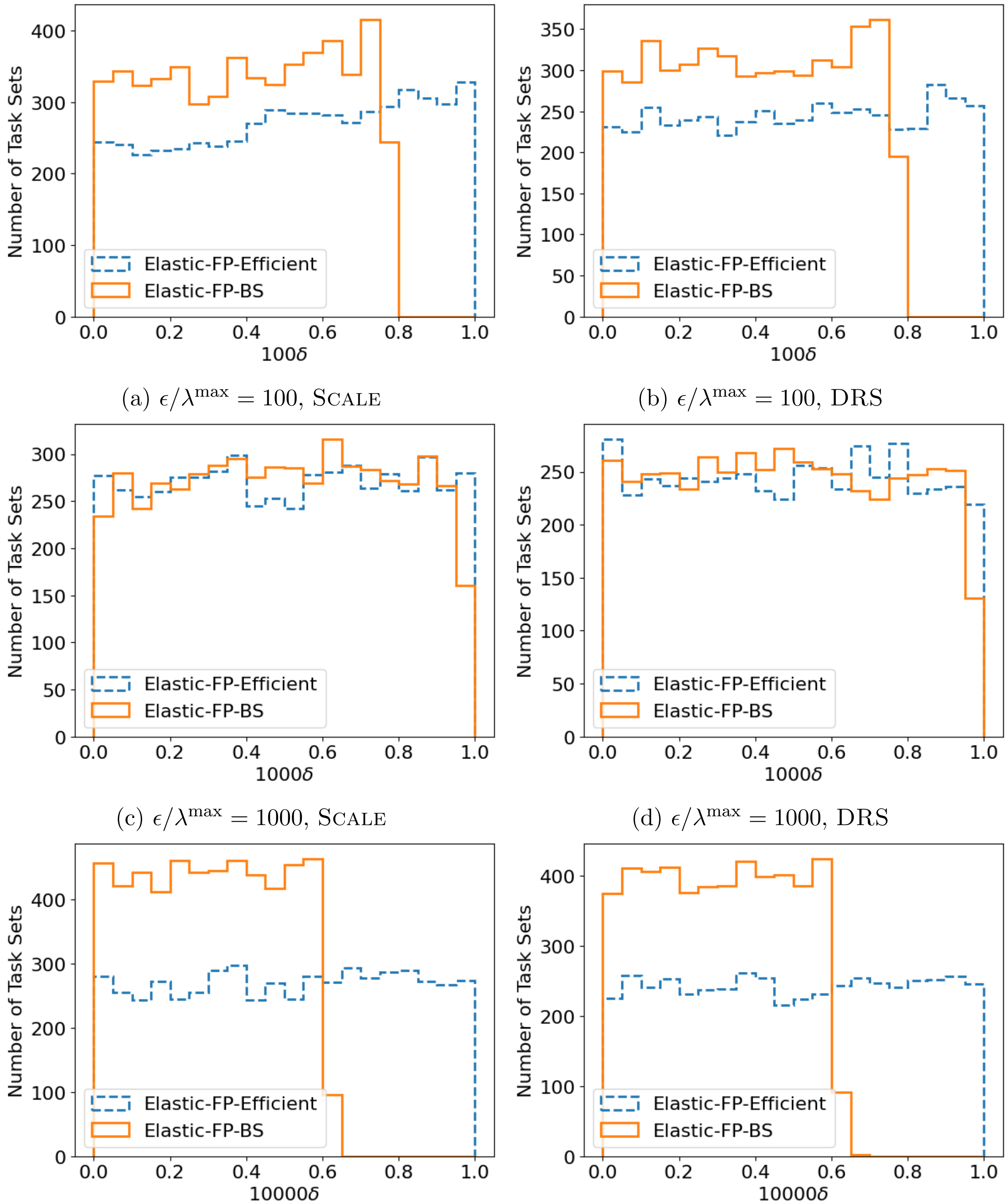
We begin by comparing the relative compression achieved by each remaining algorithm to illustrate when approximation may be sufficient. We define the metric

$$\delta = \frac{\lambda - \lambda^*}{\lambda^{\max}}$$

representing the difference between the compression value λ returned by an approximate algorithm and the optimal λ^* returned by Elastic-FP-MIQP, normalized by $\lambda^{\max}$.

We consider just those sets of 50 or fewer tasks for which elastic compression achieves a feasible configuration — i.e., those schedulable when periods are extended to their maximum values. There are 5 415 such task sets (98.5%) with minimum utilizations assigned according to Scale, and 4 882 (88.8%) for DRS; this makes sense, since the *average* total minimum utilization $U_{\text{SUM}}^{\min}$ is twice as high with DRS.

The relative distributions are plotted in Figure 4.8. To achieve the same range of values in the horizontal axes of each plot, we multiply by $\epsilon/\lambda^{\max}$, with values closer to 0 representing better agreement with λ^*. We observe that, while similar, Elastic-FP-BS ***tends to do better than*** Elastic-FP-Efficient. Since it is also more efficient for finer granularity of ϵ, this likely makes it the preferred choice between the two approximate algorithms.



(a) $\epsilon/\lambda^{\max} = 100$, SCALE

(b) $\epsilon/\lambda^{\max} = 100$, DRS

(c) $\epsilon/\lambda^{\max} = 1000$, SCALE

(d) $\epsilon/\lambda^{\max} = 1000$, DRS

(e) $\epsilon/\lambda^{\max} = 10\,000$, SCALE

(f) $\epsilon/\lambda^{\max} = 10\,000$, DRS

Figure 4.8: Relative distance from optimal of λ returned by approximate algorithms.

Comparison of Period Assignments

To quantify the effect of using an approximate algorithm to enact period compression over an elastic task set, we next compare the relative periods achieved by each algorithm. We define the metric

$$\theta_i = \frac{T_i(\lambda)}{T_i(\lambda^*)}$$

representing the ratio of a task's period when compressed by an approximate algorithm to the period under optimal compression. We compare values of θ_i between ELASTIC-FP-EFFICIENT and ELASTIC-FP-BS for each of the three values of $\epsilon/\lambda^{\max}$ considered. Results for the 164 070 total tasks for which compression was achieved and with minimum utilizations assigned according to SCALE are summarized in Table 4.6.

	ELASTIC-FP-EFFICIENT			ELASTIC-FP-BS		
θ	100	1000	10 000	100	1000	10 000
1 ($\lambda = \lambda^*$)	79696	79776	80668	79695	79754	81204
(1,1.1]	49396	76976	82566	53826	77175	82371
(1.1,2]	26806	6311	745	23993	6128	445
(2,10]	7085	876	81	5670	884	45
(10,100]	990	122	10	803	120	5
$\geq$ 100	97	9	0	83	9	0

Table 4.6: Relative overcompression of SCALE tasks by approximate algorithms.

Table 4.7 shows the same results, but for the 154 020 tasks for which compression was achieved when assigning minimum utilizations with DRS.

	ELASTIC-FP-EFFICIENT			ELASTIC-FP-BS		
θ	100	1000	10 000	100	1000	10 000
1 ($\lambda = \lambda^*$)	112125	112206	113190	112115	112223	113957
(1,1.1]	33273	40492	40682	34573	40479	39974
(1.1,2]	7389	1187	134	6348	1181	80
(2,10]	1097	120	12	870	123	9
(10,100]	122	13	2	100	12	0
$\geq$ 100	14	2	0	14	2	0

Table 4.7: Relative overcompression of DRS tasks by approximate algorithms.

For many tasks, $\theta = 1$, i.e., $\lambda = \lambda^*$. In almost all cases, this is due to the task being fully compressed, i.e., $T_i = T_i^{\min}$. We observe that ***the relative overcompression of the* Elastic-FP-Efficient *and* Elastic-FP-BS *algorithms tends to be small*** (under 10%), especially for large values of $\lambda^{\max}/\epsilon$, and especially for DRS tasks. Nevertheless, occasionally tasks are significantly overcompressed. Even for $\lambda^{\max}/\epsilon$=10 000, Elastic-FP-Efficient and Elastic-FP-BS overcompress 15 Scale task periods by over 10×; though this is only observed for 2 DRS tasks.

4.9 Conclusion

In this chapter, we have extended uniprocessor elastic scheduling to fixed-priority, constrained-deadline tasks. We have presented four algorithms. Elastic-FP-Efficient applies compression iteratively using step sizes of a tunable value ϵ. Elastic-FP-BS instead performs a binary search over the range of possible compression values, also using a precision of ϵ. The Elastic-FP-MIQP-Joint algorithm formulates the problem of finding the *exact* amount of compression to apply as a mixed integer quadratic program.

We implemented Elastic-FP-MIQP-Joint using SCIP, an open-source, off-the-shelf constraint solver. However, even when evaluating on sets of only 10 tasks, we discovered that SCIP was often unable to efficiently converge on an optimal solution. To reduce the size of the MIQP, we proposed Elastic-FP-MIQP, which solves a simplified version of the problem for a single task at a time, achieving speedups of 2–3 orders of magnitude.

We have demonstrated that ***both approximate algorithms are highly efficient***. Even for systems of 100 tasks and small values of ϵ, Elastic-FP-BS enables compression in under 20 milliseconds on a Raspberry Pi 3 Model B+. We also observed that, when compared to the optimal compression achieved by the MIQP approach, neither approximate algorithm overcompresses periods by more than 10% for 99.5% of tasks tested. However, in very rare corner cases ($<$ 0.006% of tested tasks), the iterative algorithms overcompress periods by more than 10× the optimal compression. Therefore, ***for offline scheduling decisions, solving MIQPs may still remain a better choice***. Even for task sets of size 50, the algorithm completed in under 1.5 minutes 95% of the time, though we did observe 2 task sets for which the MIQP took over an hour to finish.

Chapter 5

Harmonic Task Systems

Portions of this chapter were published as "Elastic Scheduling for Harmonic Task Systems" at RTAS 2024 [151].

5.1 Introduction

We now return to the problem of elastic scheduling for implicit-deadline tasks. This chapter presents the first work to extend the elastic scheduling model to task systems for which periods are additionally constrained to be *harmonic*. Many control systems, such as those found in robotics applications [92] and real-time hybrid simulation (RTHS) of structural integrity [121], demand harmonic rates among their constituent tasks. In applications that capture and process frames from multiple sensing devices to be aggregated in backend processing tasks, such as simultaneous localization and mapping (SLAM) [88] and real-time mobile spectrometry [167], harmonic task periods guarantee consistent temporal alignment [91]. Furthermore, task sets with harmonic periods have hyperperiods equal to the largest period [29, 130], which reduces the size of scheduling tables in time triggered systems [83] and constrains the test set in processor demand analysis [20].

The problem of selecting harmonic periods from within acceptable intervals is nontrivial. Nasri et al. [112] formalized the problem, and proposed an approach that solves the problem in time linear in the number of tasks for restricted cases but in general "can exponentially grow." In [111], Nasri and Fohler identified another restriction of the problem that can be solved in polynomial time, but they provide "no guarantee for reasonable computational complexity" in general.

In this work, we reason about the problem of assigning task periods from continuous intervals such that *(i)* the objective function defined for elastic scheduling by Chantem et al. in [44, 45] is minimized, *(ii)* periods remain harmonic, and *(iii)* schedulability is guaranteed. We call this the **harmonic elastic problem**. Toward solving the harmonic elastic problem, this chapter proves complexity results, demonstrates a tractable restriction of the problem for online adaptation to changes in available utilization, and demonstrates applicability to real-world applications.

5.1.1 Complexity Results

In Section 5.4, we prove three results about the complexity of the problem. First, we demonstrate via a reduction from integer factorization that for n tasks and a parameter k bounding the ratio of the task periods, finding a harmonic period assignment (if one exists within the allowed intervals) is unlikely to be solvable in time polynomial in n and $\log k$. Second, we nonetheless outline a *pseudo-polynomial* $\boldsymbol{\mathcal{O}(n \log n + nk^2)}$ algorithm to do so. Third, we prove that the harmonic elastic problem on a uniprocessor is at least weakly NP-hard in general, even for fixed k.

5.1.2 A Restriction for Real Systems

In Section 5.5, we propose a two-part algorithm to find an optimal solution to the harmonic elastic problem with an *a priori* order imposed over task periods. This is a natural restriction in many systems, such as multi-time stepping decomposition of a real-time hybrid simulation (RTHS) [31] for natural hazards engineering, and applications such as ORB-SLAM [110] where front-end data collection tasks capture and process frames from sensing devices which must be aggregated downstream. Our algorithm searches offline for all projected harmonic intervals (PHIs) that can be constructed within the allowed period ranges, constructing a lookup table that identifies the *optimal* PHI for each possible utilization bound.[13] Despite being exponential in $\log k$, we demonstrate that in practice this algorithm is feasible for

[13]Intuitively, for a given sequence of harmonic multipliers, the corresponding PHI describes the largest continuous range such that if the first task in period order is assigned a period from that range, all periods assigned as corresponding integer multiples remain within the allowed intervals for each task. We define PHI more formally in Definition 4 of Section 5.5.

reasonable values of n and k and the lookup table remains small. During online execution, if available utilization changes (e.g., due to processor cores going offline, interference from background processes, or the arrival of aperiodic workloads), binary search allows reassignment of task periods in time polynomial in $\log n$ and $\log k$.

5.1.3 Real-World Applications

We apply this approach to two real-world applications: the Fast Integrated Mobility Spectrometer (FIMS) [167] and the ORB-SLAM3 [42] simultaneous localization and mapping system.

FIMS, described in Section 5.7.1, performs real-time atmospheric aerosol monitoring, aggregating and synchronizing harmonic inputs with a backend matrix inversion task that translates detected particle coordinates into a particle size distribution. Towards future deployment onboard a small UAV, we run FIMS *on a single core* of the Raspberry Pi 4, using our harmonic elastic scheduling model to prevent overload. We find that, by adjusting task periods, we can guarantee temporal alignment and avoid deadline misses, even when FIMS is limited to consuming only a fraction of that core's bandwidth.

In ORB-SLAM3, dataflow and synchronization requirements impose a harmonic total ordering over the task periods. When computational resources are insufficient to guarantee completion of all tasks, frames may be dropped or desynchronized, giving increasingly erroneous results [90]. Our harmonic elastic scheduling model allows it to adjust task periods in response to changes in available CPU bandwidth when executing concurrently with other tasks on a *single core*, achieving a 10.4× reduction in relative translational error (RTE) using this approach compared to its baseline execution.

5.2 Background and System Model

5.2.1 Elastic Scheduling

The system model for Buttazzo's elastic scheduling of recurrent, implicit-deadline tasks [39, 40] can be found in Section 2.2.2. In this section, we re-introduce key background concepts necessary for the development of elastic scheduling models for harmonic task systems.

The elastic recurrent real-time workload model [39, 40] provides a framework for managing overload without suspending tasks or dropping jobs by reducing ("compressing") individual tasks' utilizations until the total utilization no longer exceeds the schedulable bound. Each task's utilization is likened to a spring whose elasticity reflects the task's ability to adapt its utilization to a lower quality of service. If the total length of the springs, placed end to end, exceeds some desired length (the utilization bound), a compressive force is applied to the system. Each spring (and corresponding utilization) is compressed *proportionally to its elasticity* until the total utilization no longer exceeds the bound, or until the individual task reaches its minimum serviceable utilization, by extending individual task periods.

This dissertation has already mentioned and explored extensions of this model to multiprocessor scheduling of sequential, implicit-deadline tasks (Chapter 3), and to EDF ([13]) and fixed priority (Chapter 4) scheduling of constrained-deadline tasks. However, some scheduling models impose constraints where the semantics of proportional compression are no longer readily applicable — e.g., under federated scheduling [94] where every parallel task executes on dedicated cores, or (as in this chapter) when periods must remain harmonic. To allow further generalization, Chantem et al. demonstrated in [44, 45] that utilizations satisfying the elastic scheduling model could be found by solving a constrained optimization problem. We have already presented the problem formulation in Chapter 4.2.1, but we restate it here:[14]

$$\min_{U_i} \quad \sum_{i=1}^{n} \frac{1}{E_i}(U_i^{\max} - U_i)^2 \tag{5.1a}$$

$$\text{s.t.} \quad \sum_{i} U_i \leq U_D \quad \text{and} \tag{5.1b}$$

$$\forall_i, \quad U_i^{\min} \leq U_i \leq U_i^{\max} \tag{5.1c}$$

[14]In this chapter, we refer to Expression 5.1a as the "elastic objective."

By modifying the schedulability constraint (Expression 5.1b), Orr et al. developed elastic frameworks for federated scheduling of parallel tasks having periods [120] and workloads [119] that may be adjusted continuously, or selected from discrete sets of candidate values [121]. In particular, Orr et al. claim that the sets of discrete utilizations may be constructed so as to guarantee harmonic period assignments [121]. This approach, however, does not allow task utilizations to be compressed over continuous ranges within the constraints of harmonicity; nor does it address the question of *how* to select candidate harmonic period values within a range that is acceptable for each task.

5.2.2 Harmonic Periods

The problem of assigning harmonic periods to task systems has been the subject of prior work. Previous studies have considered using harmonic periods to improve schedulability. It is a well-known result (see, e.g., [84]) that rate monotonic (RM) preemptive scheduling on a uniprocessor permits a utilization bound of 1 if tasks are assigned harmonic periods. Han and Tyan presented a schedulability test that takes advantage of this by mapping task periods to smaller artificial harmonic values [74], borrowing from an earlier algorithm presented by Han and Lin [73] — if the system's utilization does not exceed 1 under these artificial periods, then all tasks are schedulable at their original periods. Shih et al. used this same algorithm to schedule radar dwells in [136]. Fan and Quan demonstrated a strategy to partition fixed-priority tasks on a multiprocessor according to their harmonic relationships [62]. Min-Allah et al. proposed a utilization bound test for task sets with deadlines larger than periods that constructs artificial harmonic deadlines [107]. In these studies, tasks still execute at their originally-assigned periods, and therefore acceptable period *intervals* need not be considered.

Period *assignment* from sets of acceptable values has also been studied in prior work. Kuo and Mok studied systems where each task is assigned a *discrete* set of periods at which it may execute, and for which execution times scale with periods (so that utilization remains constant) [84]. They considered assigning periods to tasks so that the number of subsets of tasks with harmonic periods — they refer to such a subset as a "harmonic chain" — does not exceed some value h selected to guarantee RM schedulability. For $h = 1$, this guarantees that all periods are harmonic.

Assigning periods to tasks from continuous ranges so as to minimize the hyperperiod has also been studied [29, 130]. While this approach will find an assignment (if one exists) of periods to each task such that the *largest* period is an integer multiple of the others, this does not guarantee that *all* periods are harmonic.

However, many applications demand harmonic task rates for functional correctness, not just to increase the schedulable utilization bound. These include robotic control systems [92, 91], real-time hybrid simulation [31, 63, 64, 121], SLAM systems [88, 110, 42] and mobile spectrometry [167, 169]. To satisfy such requirements, we consider the problem of assigning harmonic periods to tasks from *continuous* intervals. We build off work by Nasri et al. [112], who previously studied this problem and first proposed a solution. In their model a continuous interval $I_i = [T_i^{\min}, T_i^{\max}]$ is specified for each task τ_i from which its period must be selected. Their approach orders intervals according to $T_i^{\min}$, then searches in depth-first fashion for a sequence of "projected harmonic zones" from the first interval to the last.

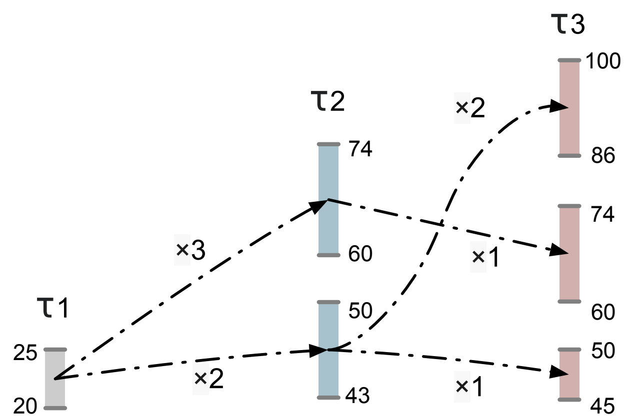


Figure 5.1: Forward search for harmonic periods via projected harmonic zones. Tasks τ_1, τ_2, and τ_3 must take periods in the intervals I_1=[20, 25], I_2=[43, 74], and I_3=[45, 100]. Projected harmonic zones from I_1 to I_2 are re-projected. Since these projections overlap I_3, harmonic periods can be assigned.

Definition 1 (Projected Harmonic Zone)**.** *(This is a restatement of [112, Definition 1].) The projected harmonic zone* $\chi^a_{I_1 \to I_2} : [T_\chi^{\min}, T_\chi^{\max}]$ *from interval* I_1 *to* I_2, $T_1^{\min} \le T_2^{\min}$*, with multiplier* $a \in \mathbb{N}^+$ *is a continuous range of numbers in* I_1 *that starts from* $T_\chi^{\min} = \max\{T_2^{\min}, a \cdot T_1^{\min}\}$ *and ends at* $T_\chi^{\max} = \min\{T_2^{\max}, a \cdot T_1^{\max}\}$.

Figure 5.1 illustrates the forward search technique proposed in [112]. Projected harmonic zones from the first interval to the second are further projected where possible to the third, and so on to the end. If any sequence reaches the last interval, then periods are selected from the corresponding zones.

Also in [112], Nasri et al. show how to identify if all harmonic zones projected onto a given interval fully overlap; in the case where this is true for all intervals, the problem is solved in linear time in the number of tasks. However, they state that in general "the number of [projected harmonic zones] can exponentially grow." In [111], Nasri and Fohler propose a backward search starting from the last interval. If all backward projections are disjoint, the problem can be solved in time $\mathcal{O}(n^2 \log n)$ for n tasks, but they provide "no guarantee for reasonable computational complexity" in general. However, in Section 5.4.2, we present a general pseudo-polynomial algorithm to assign harmonic periods from within the specified intervals.

In [108], Mohaqeqi, Nasri, et al. extend harmonic period assignment to various optimization problems. For example, they demonstrate that assigning harmonic periods from continuous intervals to minimize a weighted sum over the periods while respecting schedulability is NP-hard. We show a similar result in Section 5.4.3 for the elastic objective. In [123], Pavić and Džapo also present algorithms and complexity results for the objectives from [108], but with periods constrained to sets of discrete candidate values.

5.2.3 Other Adaptive Frameworks

Though this chapter specifically considers elastic scheduling, we also evaluate our approach in the context of another adaptive framework. An autonomous system traversing new environments must simultaneously localize and map its surroundings in real-time to maintain course and prevent collision. In such systems, sensing devices such as cameras, LiDAR, and inertial measurement units (IMUs) produce data frames which are synchronized by back-end processing tasks (e.g., feature tracking and bundle adjustment). When computational resources are insufficient to guarantee completion of all tasks, frames may be dropped or desynchronized, which can cause increasingly erroneous results.

Previous work has attempted to predict temporal budgets on systems migrated to SWaP-constrained hardware or when timing abnormalities inflate execution times past what are typical. In [90], Li et al. propose an approach to selectively drop frames to minimize relative pose error (RPE). Without selective dropping, overruns will cause frames to be lost; intentionally picking which frames to drop can improve system performance compared to "random" losses. In comparison, our model adjusts task periods in the face of insufficient computational resources to avoid dropped frames while still respecting deadlines. In Section 5.7.2, we demonstrate that this allows ORB-SLAM to adjust to background interference more effectively than the approach in [90].

5.3 Problem Statements

In this work, we consider the following three problems:

5.3.1 The Harmonic Period Problem

Given a set Γ of n tasks, where each task τ_i is characterized by a continuous interval $I_i = [T_i^{\min}, T_i^{\max}]$, assign each task a period T_i such that for all $i, 1 \leq i \leq n$, these conditions are satisfied: *(i)* $T_i \in I_i$, and *(ii)* for any i, j, either $T_i/T_j \in \mathbb{N}^+$ or $T_j/T_i \in \mathbb{N}^+$. This is the problem considered in [112, 111], and does not consider schedulability under the resulting set of assignments.

To reason about this, we introduce a parameter k that bounds the ratio of task periods:

$$k \stackrel{\text{def}}{=} \frac{\max_{\tau_i} T_i^{\max}}{\min_{\tau_i} T_i^{\min}} \tag{5.2}$$

In Section 5.4.1 we argue that the problem is unlikely to have a polynomial time solution in the number of tasks n or the size of the representation of k (even for fixed n), but in Section 5.4.2, we present a pseudo-polynomial algorithm to solve the problem.

We note that this generalizes to the problem of finding a harmonic chain of values, where each value is constrained to a continuous interval. It does not consider schedulability under the resulting set of assignments.

5.3.2 The Harmonic Elastic Problem

Given a set Γ of n implicit-deadline elastic tasks on a uniprocessor, where each task τ_i is characterized by five non-negative parameters $(C_i, T_i^{\min}, T_i^{\max}, T_i, E_i)$,[15] find an assignment of periods T_i to each task τ_i satisfying the constrained optimization problem in Expression 5.1, with the additional constraint:

$$\forall_{i,j}, \quad T_i/T_j \in \mathbb{N}^+ \text{ or } T_j/T_i \in \mathbb{N}^+ \tag{5.3}$$

In Section 5.4.3, we prove that this is NP-hard in general.

5.3.3 The Ordered Harmonic Elastic Problem

This problem is equivalent to the harmonic elastic problem, but with the additional restriction that task periods must be selected to respect some total ordering assigned a priori. In other words, $\forall i, j$ such that $1 \leq i < j \leq n$, it is required that $T_i \leq T_j$. If tasks are assigned fixed priorities and indexed accordingly, any periods thus assigned will satisfy rate monotonic scheduling. This is a natural restriction in many applications, and in Section 5.5 we show that it allows for polynomial-time online period adjustment.

5.4 Complexity Results

In this section, we prove three results about the complexity of finding harmonic periods. First, we demonstrate that in general, for a set Γ of n periodic tasks characterized as in Section 5.3.1, there is likely no algorithm polynomial in n or even in $\log k$ that can assign a set of harmonic periods T_i within the allowed intervals. Furthermore, assigning a set of harmonic periods that *minimize* the elastic objective function (5.1a) within the system's utilization bound is NP-hard.

[15] These parameters are described in Section 2.2 and Equations 4.6 and 4.7 of Section 4.2.2.

5.4.1 Complexity of the Harmonic Period Problem

The *decision version* of the harmonic period problem asks: Given a set Γ of n tasks, where each task τ_i is characterized by a continuous interval $I_i = [T_i^{\min}, T_i^{\max}]$ — from which the parameter k as defined in Equation 5.2 can be derived — *is it possible to* assign each task τ_i a period T_i such that the following conditions are satisfied: *(i)* $T_i \in I_i$, and *(ii)* for any i, j, either $T_i/T_j \in \mathbb{N}^+$ or $T_j/T_i \in \mathbb{N}^+$?

We point out that this decision problem is no harder than the problem of actually finding the periods; hence, any lower bounds on the computational complexity of this decision problem also holds for the problem of finding the periods.

We now show, by reduction from integer factorization (defined below), that it is unlikely that this decision problem can be solved in time polynomial in n and $\log k$.

Definition 2 (Integer Factorization)**.**
  INSTANCE*: Positive integers N and ℓ, $1 < \ell < N$.*
  QUESTION*: Does N have an integer factor in $[2, \ell]$?*

Integer factorization is widely believed to not be solvable by polynomial-time algorithms (assuming $\mathsf{P} \neq \mathsf{NP}$) [109]. The following theorem builds upon this belief to show that we are unlikely to be able to solve the decision version of the harmonic period problem in polynomial time:

Theorem 5. *If the decision version of the harmonic period problem can be solved in time polynomial in n and $\log k$ — where n denotes the number of tasks and k bounds the ratio of the task periods as in Equation 5.2 — then integer factorization can be solved by a polynomial-time algorithm as well.*

Proof. We can reduce integer factorization to the decision version of the harmonic period problem as follows. For any instance (N, ℓ) of integer factorization, we construct a set of three tasks $\Gamma = \{\tau_1, \tau_2, \tau_3\}$ with period intervals as follows:

$$I_1 = [1, 1] \qquad I_2 = [2, \ell] \qquad I_3 = [N, N]$$

Note that τ_1's period T_1 must equal 1 and τ_3's period T_3 must equal N. The harmonicity constraint mandates that *(i)* τ_2's period T_2 be divisible by $T_1 \equiv 1$, and hence an integer within the interval $[2, \ell]$; and *(ii)* this period T_2 must be a divisor of τ_3's period $T_3 \equiv N$. Taken together, these facts imply that the task system Γ constructed above is a YES instance for the decision version of the harmonic period problem if and only if N has an integer factor in $[2, \ell]$; i.e., if (N, ℓ) is a YES instance of integer factorization.

We have shown that integer factorization of N can be reduced to an instance of the harmonic period problem with $n = 3$ and $k = N$ as defined in Equation 5.2, and so the harmonic period problem for $n = 3$ is at least has hard as integer factorization. This result implies that ***we are unlikely to be able to obtain a polynomial-time algorithm in n or $\log k$ (even for fixed n) that solves the Harmonic Period Problem.*** □

This motivates our efforts to obtain a *pseudo*-polynomial algorithm that solves it. In the next section, we present an $O(n \log n + nk^2)$ algorithm.

5.4.2 An Algorithm for the Harmonic Period Problem

We now derive a pseudo-polynomial algorithm that solves the harmonic period problem. Intuitively, it is similar to Nasri et al.'s forward projection approach [112] (illustrated in Section 5.2.2, Figure 5.1), but we observe that if adjacent intervals I_i, I_{i+1} overlap (i.e., if $T_{i+1}^{\min} < T_i^{\max}$), then the search from projected harmonic zones into this overlapping region in I_i only requires a re-projection into I_{i+1} using the multiplier a=1, as illustrated in Figure 5.2.

Algorithm Description

The procedure is outlined in Algorithm 9. It first sorts tasks in ascending order of $T_i^{\min}$; in [112, Corollary 1], Nasri et al. showed that projecting harmonic zones in this order will find a set of harmonic periods, if one exists. It then performs a breadth-first search for projected harmonic zones over just those intervals that do not enclose any subsequent intervals, as illustrated in Figure 5.3.

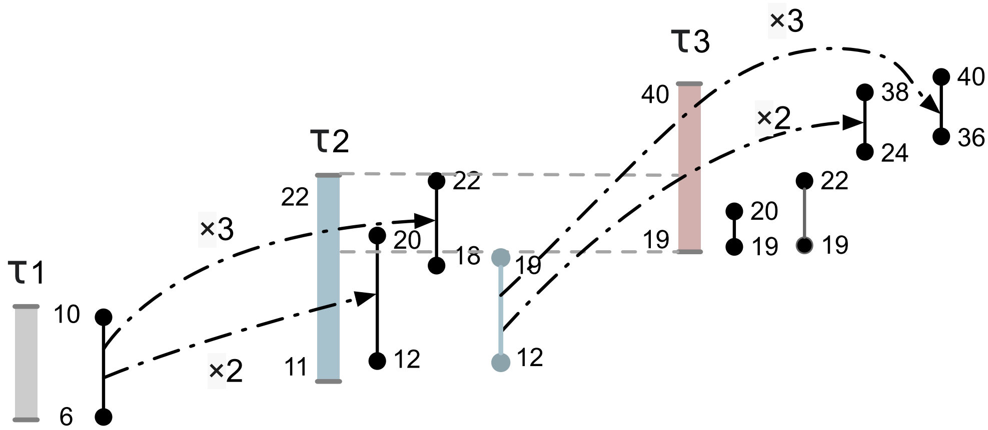


Figure 5.2: Tasks τ_1–τ_3 have intervals I_1=[6, 10], I_2=[11, 22], I_3=[19, 40]. Projected harmonic zones $\chi^2_{I_1 \to I_2}$: [12, 20] and $\chi^3_{I_1 \to I_2}$: [18, 22] both overlap I_3. The overlapping portions are re-projected using only the multiplier $a = 1$, forming $\chi^1_{I_2 \to I_3}$: [19, 20] and $\chi^1_{I_2 \to I_3}$: [19, 22]. The non-overlapping portions [12, 19] and [18, 19] are merged and re-projected into I_3 starting from multiplier $a = 2$, forming $\chi^2_{I_2 \to I_3}$: [24, 38] and $\chi^3_{I_2 \to I_3}$: [36, 40].

Algorithm 9: FIND-HARMONIC-PERIODS(Γ)

Input: A set Γ of n tasks with period intervals $I_i = [T_i^{\min}, T_i^{\max}]$
Output: A set of harmonic period assignments $\{T_i\}$

▷ `Initialize task set`
Sort Γ in ascending order of $T_i^{\min}$
$S \leftarrow \{\}$ ▷ `Sequence of period intervals`
forall $1 \leq i < n$ **do**
  if $T_i^{\max} < T_{i+1}^{\max}$ **then** Insert I_i into **S**
Insert I_n into **S**

▷ `Find projected harmonic zones`
$j \leftarrow$ first element in **S**
SOURCES $\leftarrow \{(I_j, \{0\})\}$
(INTERVAL, $\{a_i\}$) $\leftarrow$ PROJECT(**S**, SOURCES, 2)
if FAILURE **then return** FAILURE

▷ `Assign periods from harmonic zones`
Assign to T_n any value in INTERVAL
forall $i : n{-}1 \rightarrow 1$ **do**
  if $I_i \in S$ **then** $T_i = T_{i+1}/a_j$
  else $T_i = T_{i+1}$
return $\{T_i\}$

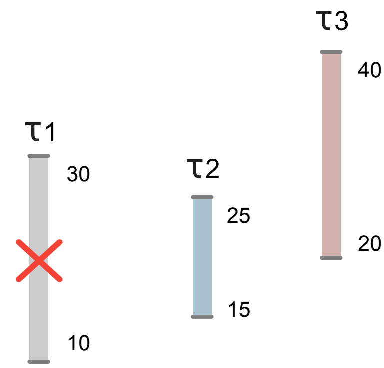


Figure 5.3: Task τ_1 has period interval I_1=$[10, 30]$ and τ_2 has I_2=$[15, 25]$. As I_1 encloses I_2, we remove τ_1 from the search space. Its period T_1 can take the value assigned to T_2. Task τ_3 has I_3=$[20, 40]$, which overlaps but is not enclosed by I_2.

The following lemma proves that enclosing intervals can be removed from the search space.

Lemma 1. *If adjacent tasks τ_i and τ_{i+1} have intervals I_i and I_{i+1} such that $T_i^{\min} \leq T_{i+1}^{\min}$ and $T_i^{\max} \geq T_{i+1}^{\max}$, we say that the interval I_i* ***encloses*** *I_{i+1}. Then if any solution to the harmonic period problem exists, there must exist a solution that satisfies $T_i = T_{i+1}$.*

Proof. Assume that for a task system Γ there exists a harmonic assignment of periods T_i to each task τ_i such that $T_i \in I_i$, i.e., for all i, j, either $T_i/T_j \in \mathbb{N}^+$ or $T_j/T_i \in \mathbb{N}^+$. Then it follows from [112, Corollary 1] that for any two tasks τ_i, τ_j for which $T_i^{\min} \leq T_j^{\min}$ there is some such assignment for which $T_j/T_i \in \mathbb{N}^+$, implying that $T_i \leq T_j$. Now assume that $T_i^{\max} \geq T_j^{\max}$. If we reassign T_i to the value T_j, it still follows that for all k, either $T_i/T_k \in \mathbb{N}^+$ or $T_k/T_i \in \mathbb{N}^+$. Additionally, since $T_i^{\min} \leq T_j^{\min} \leq T_j \leq T_j^{\max} \leq T_i^{\max}$, it still holds that $T_i \in I_i$. So the harmonic period problem is still satisfied. □

The algorithm proceeds via recursive calls to PROJECT, outlined in Algorithm 10. This takes a list (SOURCES) of projected harmonic zones in the current interval to be re-projected to the next interval, each paired with the sequence of multipliers that produced it. It starts

Algorithm 10: PROJECT($\mathbf{S}$, SOURCES, i)

Inputs: A set $\mathbf{S}$ of continuous period intervals $I_i = [T_i^{\min}, T_i^{\max}]$
A set SOURCES of projected harmonic zones $[T^{\min}, T^{\max}]$ and corresponding sequence of multipliers $\mathbf{M}$
An index i of the period interval in $\mathbf{S}$ into which to project
Outputs: A projected harmonic zone into the last interval in $\mathbf{S}$
The sequence of integer multipliers that produces the corresponding set of projected harmonic zones

▷ `Reached the end`
if $i \geq |\mathbf{S}|$ **then**
    if SOURCES *is not empty* **then** return any element in SOURCES
    else return FAILURE

$j \leftarrow i^{\text{th}}$ interval in $\mathbf{S}$
TARGETS $\leftarrow \{\}$
$o \leftarrow T_j^{\min}$; $k \leftarrow -1$; ▷ `Track lower-bound and index of largest non-overlapping split region`
forall $([T_s^{\min}, T_s^{\max}], M) \in$ SOURCES **do**
    ▷ `Source zone overlaps target`
    if $T_j^{\min} < T_s^{\max}$ **then**
        Insert 1 into $\mathbf{M}$
        Insert $([T_j^{\min}, T_s^{\max}], \mathbf{M})$ into TARGETS
        if $T_s^{\min} < o$ **then** $o \leftarrow T_s^{\min}$; $k \leftarrow s$
    ▷ `No overlap`
    else
        $a^{\min} \leftarrow \lceil T_j^{\min} / T_s^{\max} \rceil$; $a^{\max} \leftarrow \lfloor T_j^{\max} / T_s^{\min} \rfloor$
        forall $a^{\min} \leq a \leq a^{\max}$ **do**
            $\mathbf{M}' \leftarrow \mathbf{M}$
            Insert a into $\mathbf{M}'$
            $T^{\min} \leftarrow \max\{T_j^{\min}, a \cdot T_s^{\min}\}$
            $T^{\max} \leftarrow \min\{T_j^{\max}, a \cdot T_s^{\max}\}$
            Insert $([T^{\min}, T^{\max}], \mathbf{M}')$ into TARGETS

▷ `There was overlap`
if $k > -1$ **then**
    $T^{\min} \leftarrow o$; $T^{\max} \leftarrow T_j^{\min}$; $a^{\max} \leftarrow \lfloor T_j^{\max} / T^{\min} \rfloor$;
    forall $2 \leq a \leq a^{\max}$ **do**
        $\mathbf{M}' \leftarrow k^{\text{th}}$ multiplier sequence in $\mathbf{S}$
        Insert a into $\mathbf{M}'$ $T^{\min} \leftarrow \max\{T_j^{\min}, a \cdot T_s^{\min}\}$
        $T^{\max} \leftarrow \min\{T_j^{\max}, a \cdot T_s^{\max}\}$
        Insert $([T^{\min}, T^{\max}], \mathbf{M}')$ into TARGETS
return PROJECT($\mathbf{S}$, SOURCES, $i + 1$)

with the complete first interval and terminates when the final interval is reached. To enable breadth-first search, it keeps a list (TARGETS) of projected harmonic zones into the next interval. For all projected harmonic zones in SOURCES, the algorithm first checks if the zone overlaps the next interval. If it does not, all re-projected harmonic zones (and their corresponding multiples) from the source zone into the next interval are added to TARGETS. However, if it does, the zone is split (if necessary) into overlapping and non-overlapping regions (see Figure 5.2). The overlapping region (with multiplier 1) is added to TARGETS.

The following lemma argues for the correctness of PROJECT's handling of overlapping regions.

Lemma 2. *If adjacent tasks τ_i and τ_{i+1} have intervals such that $T_i^{\min} \leq T_{i+1}^{\min}$, $T_i^{\max} > T_{i+1}^{\min}$, and $T_i^{\max} < T_{i+1}^{\max}$, and if a solution to the harmonic period problem exists where T_i is assigned some value t such that $T_{i+1}^{\min} \leq t$, then a solution must exist where both $T_i = t$ and $T_{i+1} = t$.*

Proof. Assume that for a task system Γ there exists a harmonic assignment of periods T_i to each task τ_i such that $T_i \in I_i$, i.e., for all i, j, either $T_i/T_j \in \mathbb{N}^+$ or $T_j/T_i \in \mathbb{N}^+$. Then it follows from [112, Corollary 1] that for any two tasks τ_i, τ_j for which $T_i^{\min} \leq T_j^{\min}$ there is some such assignment for which $T_j/T_i \in \mathbb{N}^+$, implying that $T_i \leq T_j$. Now assume that $T_i^{\max} > T_j^{\min}$ and $T_i^{\max} < T_j^{\max}$. If we reassign T_j to the value T_i, it still follows that for all k, either $T_j/T_k \in \mathbb{N}^+$ or $T_k/T_j \in \mathbb{N}^+$. Additionally, since $T_j^{\min} \leq T_j \leq T_i^{\max} \leq T_j^{\max}$, it still holds that $T_{i+1} \in I_{i+1}$. So the harmonic period problem is still satisfied. □

This says that the portion of a projected harmonic zone χ in I_i containing periods that would be contained within the next interval I_{i+1} does not have to be projected into I_{i+1} using multipliers greater than 1 if $T_{i+1}^{\max} > T_\chi^{\max}$. Because we have eliminated from the search those intervals where $T_{i+1}^{\max} < T_i^{\max}$, this condition holds.

After all SOURCES have been re-projected into the next interval as described above, it remains to deal with the largest among the non-overlapping regions that have been split off from overlapping regions, if there are any (all such split regions are subsets of the largest). Projections from this region are added to TARGETS; these start from multiplier 2, since the overlapping region has already been projected with a multiplier of 1. The procedure is then called recursively for the next interval, with TARGETS passed as the new list of SOURCES.

If PROJECT succeeds in finding harmonic zones projected into the last interval, it returns one with its sequence of multiples to the calling procedure. Any value in the projected harmonic zone may be assigned as the period T_n of the last task τ_n, then other task periods are obtained by dividing by their corresponding multipliers. Any task τ_i with an interval completely enclosing the next (and that was therefore not included in the search) is assigned a period $T_i{=}T_{i+1}$ per Lemma 1.

Execution Time

For n tasks and $k = T_n^{\max}/T_1^{\min}$, this algorithm executes in time $\boldsymbol{\mathcal{O}(n\log n{+}nk^2)}$, according to the following steps.

1. Sort intervals and eliminate from the search those which fully enclose subsequent intervals: $\mathcal{O}(n \log n)$.
2. Recursively find projected harmonic zones: $\mathcal{O}(nk^2)$.
3. Assign periods from harmonic zones: $\mathcal{O}(n)$.

Steps ① and ③ are obvious, but the recursive procedure of step ② requires further analysis. We begin by showing that if intervals do *not* overlap, then the number of projected harmonic zones into the last interval does not exceed k^2.

Theorem 6. *Assume that a set of n intervals $\{I_i\}$ are ordered in such a way that $\forall i, j$, if $1{\le}i{<}j{\le}n$, then $T_i^{\min}{\le}T_j^{\min}$. Assume further that intervals do not overlap, i.e., $T_i^{\max}{\le}T_j^{\min}$. Then there can be at most k^2 projected harmonic zones into the last interval I_n.*

Proof. For any i, $2{<}i{\le}n$, the number of projected harmonic zones into I_i from a harmonic zone in I_{i-1} cannot exceed $\lfloor T_i^{\max}/T_{i-1}^{\min}\rfloor$. This follows from [112, Lemma 1].

Then the number of projected zones into I_n cannot exceed

$$\left\lfloor \frac{T_2^{\max}}{T_1^{\min}} \right\rfloor \left\lfloor \frac{T_3^{\max}}{T_2^{\min}} \right\rfloor \cdots \left\lfloor \frac{T_n^{\max}}{T_{n-1}^{\min}} \right\rfloor \le \frac{T_2^{\max}}{T_1^{\min}} \cdot \frac{T_3^{\max}}{T_2^{\min}} \cdot \ldots \cdot \frac{T_n^{\max}}{T_{n-1}^{\min}}$$

Then since for all $1 \le i < n$, $T_i^{\max} \le T_{i+1}^{\min}$, we have:

$$\frac{T_2^{\max}}{T_1^{\min}} \cdot \frac{T_3^{\max}}{T_2^{\min}} \cdot \ldots \cdot \frac{T_n^{\max}}{T_{n-1}^{\min}} \le \frac{T_2^{\max}}{T_1^{\min}} \cdot \frac{T_3^{\max}}{T_1^{\max}} \cdot \frac{T_4^{\max}}{T_2^{\max}} \cdot \ldots \cdot \frac{T_n^{\max}}{T_{n-2}^{\max}} = \frac{T_{n-1}^{\max} \cdot T_n^{\max}}{T_1^{\min} \cdot T_1^{\max}} \le \left(\frac{T_n^{\max}}{T_1^{\min}}\right)^2 = k^2 \quad \square$$

We next show that if intervals *do* overlap, the number of projected zones searched by the algorithm does not increase.

Theorem 7. *Assume that a set of n intervals $\{I_i\}$ are ordered in such a way that $\forall i, j$, $1 \leq i < j \leq n$, $T_i^{\min} \leq T_j^{\min}$ and $T_i^{\max} < T_j^{\max}$, but that some intervals overlap, i.e., for some i, $T_i^{\max} > T_{i+1}^{\min}$. Nonetheless, there are still at most k^2 harmonic zones projected into the last interval I_n.*

Proof. Consider adjacent intervals I_i, I_{i+1}. We define w_i, the number of projected harmonic zones into I_i. Then $w_i = x_i + y_i$, where x_i is the number of projected harmonic zones at least partially overlapping I_{i+1} and y_i is the number not overlapping. If $x_i = 0$ then the number of projections from I_i to I_{i+1} is the same as if the intervals themselves do not overlap. Otherwise, if $x_i > 0$, then the number of projected harmonic zones into I_{i+1} is no more than:

$$x_i + (y_i + 1) \cdot \left(\left\lfloor \frac{T_{i+1}^{\max}}{T_i^{\min}} \right\rfloor - 1 \right) \tag{5.4}$$

The x_i term is due to the overlapping regions which are projected only once with a multiplier of 1. The largest non-overlapping region split off from the projected harmonic zones that partially overlap I_{i+1}, as well as the y_i projected harmonic zones that do not overlap, contribute to the term $(y_i + 1)$. These must be re-projected into I_{i+1} (see Figure 5.2). However, because those projected harmonic zones χ that do not overlap I_{i+1} have $T_\chi^{\max} < I_{i+1}^{\min}$, the projected harmonic zones from these regions into I_{i+1} have multiples no less than 2. Similarly (per line 31 of Algorithm 10) projections from the largest split region may begin from 2, because the value $I_{i+1}^{\min}$ in the corresponding harmonic chain is already captured by the projection of the overlapping region with a multiplier of 1. Thus, it follows that the number of projected harmonic zones from these regions cannot exceed $\left(\lfloor T_{i+1}^{\max} / T_i^{\min} \rfloor - 1 \right)$. Then we define $z_i = \lfloor T_{i+1}^{\max} / T_i^{\min} \rfloor$, so the number of projected harmonic zones does not exceed:

$$x_i + (y_i + 1) \cdot (z_i - 1) = x_i + y_i z_i + z_i - y_i - 1$$

$$\leq x_i + z_i - 1 - x_i z_i + x_i z_i + y_i z_i = -(x_i - 1)(z_i - 1) + x_i z_i + y_i z_i$$

Then since $T_i^{\min} < T_{i+1}^{\max}$, it follows that $z_i \geq 1$. And since $x_i \geq 1$, it follows that $(x_i - 1)(z_i - 1) \geq 0$, so:

$$-(x_i - 1)(z_i - 1) + x_i z_i + y_i z_i \leq x_i z_i + y_i z_i = (x_i + y_i) z_i$$

This is precisely $w_i \cdot \lfloor T_{i+1}^{\max}/T_i^{\min} \rfloor$, which upper bounds the number of projected harmonic zones into I_{i+1} if it does not overlap I_i. Thus, overlapping regions do not increase the number of projected harmonic zones. $\square$

Then, since there can be no more than k^2 projected harmonic zones into any single interval, it follows that there can be no more than $(n-1)k^2$ projected harmonic zones across n intervals. So the described algorithm runs in time $\mathcal{O}(nk^2)$.

5.4.3 Complexity of the Harmonic Elastic Problem

Even if a set of assignments is found that satisfies the harmonic period problem, finding the assignment that satisfies the harmonic *elastic* problem is still at least weakly NP-hard, even for a fixed value of k bounding the ratio of the task periods as in Equation 5.2. A similar result is shown in [108, Section 3.3] for the "cost-minimizing harmonic period assignment" (CHPA) problem. In CHPA, objective (5.1a) is replaced with minimization over a weighted sum of periods; otherwise, it is equivalent to the harmonic elastic problem. We therefore prove this similarly to the approach of [108, Theorem 3] by showing a polynomial time transformation from any given instance of integer partitioning to an instance of the harmonic elastic problem.

Definition 3 (The Partitioning Problem)**.** *Let A=$\{s_1, \ldots, s_n\}$ be a set of positive integers. The problem is to determine whether A can be partitioned into two sets A_1 and A_2 such that the sum of integers in A_1 equals that of A_2. That is, if:*

$$S_1 = \sum_{s_i \in A_1} s_i \quad \textit{and} \quad S_2 = \sum_{s_i \in A_2} s_i \tag{5.5}$$

then the problem is to decide whether there exist sets A_1 and A_2 such that $A_1 \cup A_2 = A$, $A_1 \cap A_2 = \emptyset$, and $S_1 = S_2$.

Consider an arbitrary instance of the partitioning problem over n positive integers. We construct a corresponding instance of the harmonic elastic problem consisting of a set Γ of $n+1$ elastic tasks scheduled with a utilization bound $U_D = 1$. For each task τ_i, $i \leq n$, we assign as its worst-case execution time

$$C_i = \frac{4s_i}{3S} \tag{5.6}$$

We allow task periods to be assigned in the interval $[1, 2]$, i.e., for each task τ_i, $i \leq n$, $I_i = [1, 2]$. Elasticity constants are assigned to each task τ_i, $i \leq n$ as $E_i = C_i/4$. The remaining task, τ_{n+1} is assigned $C_{n+1} = 0$ and $I_i = [1, 1]$, making it inelastic. Thus, the parameter k that bounds the ratio of task periods is fixed to 2. Now, we let $\{T_1, T_2, \ldots, T_{n+1}\}$ denote a possible period assignment, and define a value O corresponding to the value taken by objective (5.1a) for this assignment:

$$O = \sum_{i=1}^{n+1} \frac{1}{E_i} (U_i^{\max} - U_i)^2 \tag{5.7}$$

We also define a value O^* denoting the minimum (i.e., optimal) value of O for which constraints (5.1b), (5.1c), and (5.3) are met.

Lemma 3. *For the above problem instance, $O^* \geq 2/3$.*

Proof. Let T_i denote the period of task τ_i assigned by a solution to the harmonic elastic problem. According to the specified task parameters, the period T_{n+1} is 1, so all other tasks must take periods of either 1 or 2 to remain harmonic. We then define the sets $\mathbb{T}_1 = \{\tau_i | T_i = 1\}$ and $\mathbb{T}_2 = \{\tau_i | T_i = 2\}$. We similarly define the terms: $\mathbb{C}_1 = \sum_{\tau_i \in \mathbb{T}_1} C_i$, $\mathbb{C}_2 = \sum_{\tau_i \in \mathbb{T}_2} C_i$, and $\mathbb{C} = \sum_i C_i$, from which it follows that $\mathbb{C} = \mathbb{C}_1 + \mathbb{C}_2$ and

$$\mathbb{C} = \sum_{i=1}^{n} \frac{4s_i}{3S} = \frac{4}{3S} \sum_{i=1}^{n} s_i = \frac{4}{3S} \cdot S = \frac{4}{3} \tag{5.8}$$

Then total utilization is:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} = \mathbb{C}_1 + \frac{\mathbb{C}_2}{2} \tag{5.9}$$

It also follows that, since for all $\tau_i \in \mathbb{T}_1$, $U_i = U_i^{\max}$, we can express O as:

$$O = \sum_{\tau_i \in \mathbb{T}_2} \frac{1}{E_i}(U_i^{\max} - U_i)^2 = \sum_{\tau_i \in \mathbb{T}_2} \frac{4}{C_i}(C_i - \frac{C_i}{2})^2 = \sum_{\tau_i \in \mathbb{T}_2} C_i = \mathbb{C}_2 \tag{5.10}$$

Then from (5.9) it follows that:

$$2\mathbb{C} - 2U = 2\mathbb{C}_1 + 2\mathbb{C}_2 - 2\mathbb{C}_1 - \mathbb{C}_2 = \mathbb{C}_2 = O \tag{5.11}$$

Equivalently, $O = 2\mathbb{C} - 2U$. Then from (5.8) it follows that:

$$O = 2(4/3) - 2U = 8/3 - 2U \tag{5.12}$$

Since U is constrained as in (5.1b) to $U \leq U_D$, we have $O \geq 8/3 - 2 = 2/3$, and so it follows that $O^* \geq 2/3$. □

Lemma 4. *A given instance of the partitioning problem is positive if and only if $O^* = 2/3$ in the constructed instance of the harmonic elastic problem.*

Proof. We first show that if the given partitioning problem is a positive instance, then the optimal period assignment satisfies $O^* = 2/3$. Let A_1 and A_2 denote the partitions of A for which $S_1 = S_2$. We assign task periods such that $T_i = 1$ if $s_i \in A_1$ and $T_i = 2$ if $s_i \in A_2$. It then follows from the assignments of task WCETs in (5.6) that $\mathbb{C}_1 = \mathbb{C}_2 = \mathbb{C}/2$. Since from (5.8) we have $\mathbb{C} = 4/3$, it follows that $\mathbb{C}_2 = 2/3$ and so from (5.10), $O = 2/3$. Since from the previous lemma, we have $O^* \geq 2/3$, it follows that $O^* = 2/3$.

Next, we show that if $O^* = 2/3$, then the given partitioning problem is a positive instance. If $O^* = 2/3$, then (5.10) implies that $\mathbb{C}_2 = 2/3$. Also, since from (5.8), $\mathbb{C} = 4/3$, this implies that $\mathbb{C}_1 = 2/3$. Since $\mathbb{C}_1 = \mathbb{C}_2$, this implies that:

$$\sum_{\tau_i \in \mathbb{T}_1} \frac{4s_i}{3S} = \sum_{\tau_i \in \mathbb{T}_2} \frac{4s_i}{3S} \tag{5.13}$$

or equivalently, $\sum_{\tau_i \in \mathbb{T}_1} s_i = \sum_{\tau_i \in \mathbb{T}_2} s_2$, which implies that $S_1 = S_2$, so the problem is positive. □

Together, the above lemmas prove that ***the harmonic elastic problem is at least weakly*** `NP`***-hard even for fixed k.***

5.5 The Ordered Harmonic Elastic Problem

We restrict our attention now to the ordered harmonic elastic problem, as defined in Section 5.3.3.

5.5.1 Preliminaries

Suppose we have a set Γ of n tasks, indexed so that for any τ_i, τ_j, if $i < j$ then $T_i \leq T_j$. It follows, then, that an assignment of harmonic periods can be expressed according to T_1 (the shortest period assignment) and a set of multiples $\{a_i, 1{\leq}i{\leq}n\}$ where $T_i = a_i \cdot T_1$. Then $a_1{=}1$ and for all i, $a_i \in \mathbb{N}^+$.

Definition 4 (Projected Harmonic Interval (PHI))**.** *For a set Γ of n elastic tasks, a projected harmonic interval is an ordered collection of values $P_j = (T_j^{\min}, T_j^{\max}, a_{1,j}, \ldots, a_{n,j})$ with $a_{1,j}{\equiv}1$. It represents the largest continuous interval $[T_j^{\min}, T_j^{\max}] \subseteq [T_1^{\min}, T_1^{\max}]$ from which a period T_1 can be selected so that a period $T_i = a_{i,j} \cdot T_1$ assigned to task τ_i is within the interval I_i that characterizes the task. Furthermore, for all b, c, $1 \leq b \leq c \leq n$ it holds that $a_{c,j}/a_{b,j} \in \mathbb{N}^+$.*

Put simply, for a given chain of harmonic multipliers, the corresponding projected harmonic interval describes the largest continuous range such that if τ_1 is assigned a period from that range, all periods assigned as corresponding integer multiples remain within the allowed intervals for each task.

5.5.2 Enumeration-Based Solution Approach

The set of all PHIs $\mathbf{P} = \{P_j\}$ for a task set represents the complete space of joint harmonic period assignments that can be selected from the continuous intervals characterizing each task. Thus, a naïve approach to optimally solving the ordered harmonic elastic problem for a given utilization bound is to enumerate the complete set of PHIs, then determine which PHI minimizes the elastic objective (Expression 5.1a) for that utilization. To do so, we define $U_{P_j}^{\min}$, the minimum utilization bound that can accommodate PHI P_j. This can be calculated as:

$$U_{P_j}^{\min} = \sum_i \frac{C_i}{a_{i,j} \cdot T_j^{\max}} = \frac{1}{T_j^{\max}} \sum_i \frac{C_i}{a_{i,j}} \tag{5.14}$$

For simplicity of notation, we define the terms $y_{i,j} = C_i/a_{i,j}$ and $Y_j = \sum_i y_{i,j}$. Then

$$U_{P_j}^{\min} = Y_j/T_j^{\max}$$

Similarly, the maximum utilization that can be achieved by PHI P_j is

$$U_{P_j}^{\max} = Y_j/T_j^{\min}$$

Even if a greater utilization bound can be accommodated, the task set executes with a utilization of $U_{R_j}^{\min}$ if periods are set according to the PHI P_j. It follows from Expression 5.1a that for $U_D \geq U_{P_j}^{\min}$, the elastic objective $O_{P_j}(U_D)$ for PHI P_j is:

$$O_{P_j}(U_D) = \begin{cases} \sum_{i=1}^{n} \frac{1}{E_i} \left(U_i^{\max} - \frac{y_{i,j} U_D}{Y_j} \right)^2 & \text{if } U_D \leq U_{P_j}^{\max} \\ \sum_{i=1}^{n} \frac{1}{E_i} \left(U_i^{\max} - \frac{y_{i,j} U_{P_j}^{\max}}{Y_j} \right)^2 & \text{if } U_D > U_{P_j}^{\max} \end{cases} \tag{5.15}$$

Our naïve approach computes $O_{P_j}(U_D)$ for every PHI P_j. The PHI that minimizes the objective is selected, and task τ_1 is assigned the period $T_{1,j}^{U_D}$:

$$T_{1,j}^{U_D} = \min \left\{ T_j^{\min}, \frac{Y_j}{U_D} \right\} \tag{5.16}$$

Other task periods are then assigned as $T_i = a_{i,j} \cdot T_{1,j}^{U_D}$.

5.5.3 Bounding Enumeration

This enumeration-based approach may be inefficient as the number of PHIs grows rapidly. However, this growth is bounded, a result which we will use to provide a polynomial-time algorithm for online adjustment.

Theorem 8. *For n tasks and $k{=}T_n^{\max}/T_1^{\min}$, the number of PHIs $|\mathbf{P}|$ is bounded by $k(n-1)^{\lfloor \log k \rfloor}$.*

Proof. We know that there is a sequence $\{a_2, \ldots, a_n\}$, $a_i \in \mathbb{N}^+$ such that

$$T_1^{\min} \cdot a_2 \cdot \ldots \cdot a_n \leq T_n^{\max}$$

which implies that

$$a_2 \cdot \ldots \cdot a_n \leq T_n^{\max}/T_1^{\min} = k$$

We define $h_{n,k}$ as the number of unique sequences $\{a_2, \ldots, a_n\}$ satisfying:

$$a_2 \cdot \ldots \cdot a_n \leq k$$

Similarly, we define $h^*_{n,k}$ as the number of unique sequences $\{a_2, \ldots, a_n\}$ satisfying:

$$a_2 \cdot \ldots \cdot a_n = k$$

Then it follows that

$$h_{n,k} = \sum_{\ell=1}^{k} h^*_{n,\ell}$$

We say an integer ℓ has a unique factorization into p_ℓ primes. Then a sequence satisfying $a_2 \cdot \ldots \cdot a_n = \ell$ must be formed in such a way that each value a_i, $2{<}i{\leq}n$ is the product of 1 and zero or more of the prime factors of ℓ, selected without replacement. The number of unique assignments of p_ℓ primes to $n{-}1$ factors is exactly $(n-1)^{p_\ell}$ for non-repeated prime factors, fewer otherwise. And the value p_ℓ is upper bounded by $\lfloor \log \ell \rfloor$. So

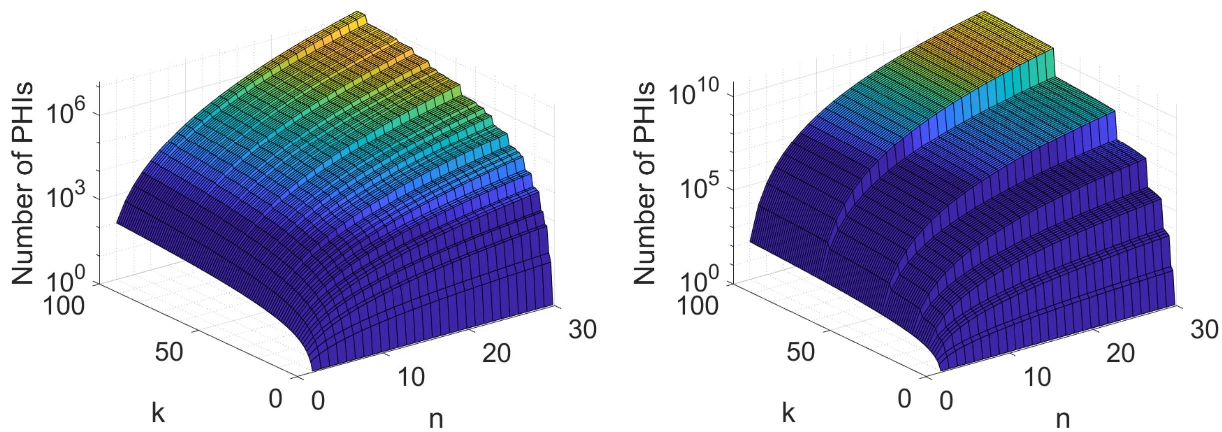
$$h^*_{n,\ell} \leq (n-1)^{\lfloor \log \ell \rfloor}$$

and so

$$h_{n,k} = \sum_{\ell=1}^{k} h^*_{n,\ell} \leq \sum_{\ell=1}^{k} (n-1)^{\lfloor \log \ell \rfloor} \leq k(n-1)^{\lfloor \log k \rfloor}$$

□

We note that this derived upper bound is not tight, because if $\ell \neq 2^a$ for $a \in \mathbb{N}$, then $p_\ell < \lfloor \log \ell \rfloor$. Furthermore, many integers have repeated prime factors. To better analyze this bound, we compute $h_{n,k}$ for values of n from 2–30 and values of k from 1–100. Results are plotted in Figure 5.4a, and compared in Figure 5.4b with the upper bound from Theorem 8. For n=30 and k=100, the upper bound is $\sim$ 5.9e10, whereas the real count is under 1.5e7, almost 4000× fewer.



(a) Calculated number of PHIs.

(b) Upper bound on PHIs.

Figure 5.4: Enumeration of projected harmonic intervals.

5.5.4 Polynomial Online Adjustment

We now describe a more efficient algorithm to reassign task periods according to the ordered harmonic elastic problem in response to runtime changes to available utilization. It assigns the same periods as the enumeration-based approach described above, finding an optimal solution to the ordered harmonic elastic problem if one exists. Through offline enumeration of **P** — the complete set of PHIs — it constructs a lookup table over the space of utilizations that can accommodate the task system. It associates ranges of utilization with the PHI achieving the lowest elastic objective. This enables polynomial-time online period selection via binary search.

Algorithm Description

The procedure, outlined in Algorithm 11, takes a set Γ of n elastic tasks, as well as the set $\mathbf{P}$ of all PHIs. It then creates a list $\mathbf{R}$ sorted over contiguous and continuous regions of utilization, where each region $R_k = (R_k^{\min}, R_k^{\max}, P_j)$ captures the PHI P_j that achieves the lowest elastic objective for utilizations in $[R_k^{\min}, R_k^{\max}]$. To do so, it creates a region for the first PHI, then iterates over all remaining PHIs P_j. Each one is compared to all existing regions in order; those for which $R_k^{\max} \leq U_{P_j}^{\min}$ are skipped.

When comparing a PHI P_j to a region R_k, the procedure finds points within the region where it may need to be split, i.e., those distinct sub-regions where the elastic objective value achieved by P_j may be less than that of the region's PHI P_k; these sub-regions will become new regions associated with P_j. Split points are of two types. In the first considered region for which $R_k^{\max}{>}U_{P_j}^{\min}$, if $R_k^{\min}{<}U_{P_j}^{\min}$, then it splits at $U_{P_j}^{\min}$. It may also split at those points U where the elastic objectives intersect, i.e., where $O_{P_j}(U){=}O_{P_k}(U)$.

Because we are concerned only with *minimizing* the objective, we eliminate constant terms:

$$O^*_{P_j}(U_D) = \begin{cases} A_{P_j}U_D^2 - B_{P_j}U_D & \text{if } U_D \leq U_{P_j}^{\max} \\ A_{P_j}(U_{P_j}^{\max})^2 - B_{P_j}U_{P_j}^{\max} & \text{if } U_D > U_{P_j}^{\max} \end{cases} \tag{5.17}$$

where

$$A_{P_j} = \frac{1}{Y_j^2}\sum_i \frac{y_{i,j}^2}{E_i} \tag{5.18}$$

$$B_{P_j} = \frac{2}{Y_j}\sum_i \frac{y_{i,j}U_i^{\max}}{E_i} \tag{5.19}$$

Algorithm 11: GENERATE-LOOKUP-TABLE($\Gamma, \mathbf{P}$)

Inputs: A set Γ of n elastic tasks, the set $\mathbf{P}$ of all PHIs over Γ
Output: A sorted list $\mathbf{R}$ over continuous, contiguous regions $R_k = (R_k^{\min}, R_k^{\max}, P_k)$ indicating that the projected harmonic interval P_k achieves the minimum elastic objective in the utilization range $[R_k^{\min}, R_k^{\max}]$

$\triangleright$ `Compute PHI parameters`
forall $P_j \in \mathbf{P}$ **do**
  Compute $U_{P_j}^{\min}$, $U_{P_j}^{\max}$, Y_{P_j}, A_{P_j}, B_{P_j}, $O_{P_j}^{*}(U_{P_j}^{\max})$ according to Eqns. 5.14, 5.17, 5.18, 5.19

$\triangleright$ `Construct regions`
$\mathbf{R} \leftarrow \{(U_1^{\min}, \infty, P_1)\}$
forall $P_j \in \mathbf{P}, j > 1$ **do**
  forall $R_k \in \mathbf{R}$ **do**
    $(R_k^{\min}, R_k^{\max}, P_k) \leftarrow R_k$
    if k *is* 1 *and* $U_j^{\min} < R_k^{\min}$ **then**
      Insert $(U_j^{\min}, R_k^{\min}, P_j)$ into $\mathbf{R}$ before R_k
    if $U_j^{\max} \leq R_k^{\min}$ **then continue**
    if $R_k^{\min} < U_j^{\min}$ **then**
      Insert $(R_k^{\min}, U_j^{\min}, P_k)$ into $\mathbf{R}$ before R_k
      $R_k^{\min} \leftarrow U_j^{\min}$

    $\triangleright$ `Parabola/parabola intersection`
    if $A_j \neq A_k$ *and* $B_j \neq B_k$ **then** $q \leftarrow (B_j - B_k)/(A_j - A_k)$
    else $q \leftarrow 0$
    if $q \notin R_k$ *or* $q > U_j^{\max}$ *or* $q > U_k^{\max}$ **then** $q \leftarrow 0$

    $\triangleright$ `Line/parabola intersection`
    if $O_j^{\min} < O_k^{\min}$ **then**
      $\ell \leftarrow \frac{B_j - \sqrt{B_j^2 + 4A_j O_k^{\min}}}{2A_j}$ $\triangleright$ `Per Equation 5.21`
      if $\ell \notin R_k$ *or* $\ell < U_k^{\max}$ *or* $\ell > U_j^{\max}$ **then** $\ell \leftarrow 0$
    else
      $\ell \leftarrow \frac{B_k - \sqrt{B_k^2 + 4A_k O_j^{\min}}}{2A_k}$ $\triangleright$ `Per Equation 5.21`
      if $\ell \notin R_k$ *or* $\ell < U_j^{\max}$ *or* $\ell > U_k^{\max}$ **then** $\ell \leftarrow 0$

    $\triangleright$ `Split region`
    REGIONS $\leftarrow R_k$
    if $q > 0$ **then** Split R_k at q
    if $\ell > 0$ **then** Split R_k at ℓ
    Replace R_k with REGIONS

    $\triangleright$ `Associate region with better PHI`
    forall $R_\ell \in$ REGIONS **do**
      $U_\ell \leftarrow (R_\ell^{\min} + R_\ell^{\max})/2$
      $O_k \leftarrow \max\left\{O_\ell^{\min}, A_k \cdot U_\ell^2 - B_k \cdot U_\ell\right\}$
      $O_j \leftarrow \max\left\{O_j^{\min}, A_j \cdot U_\ell^2 - B_j \cdot U_\ell\right\}$
      if $O_j < O_i$ **then** R_ℓ points to P_j
      $\triangleright$ `Merge redundant regions`
      if P_ℓ *matches* $P_{\ell-1}$ **then**
        $R_\ell^{\min} \leftarrow R_{\ell-1}^{\min}$
        Delete $R_{\ell-1}$

return $\mathbf{R}$

Then there are two points where the objectives may intersect, as illustrated in Figure 5.5.

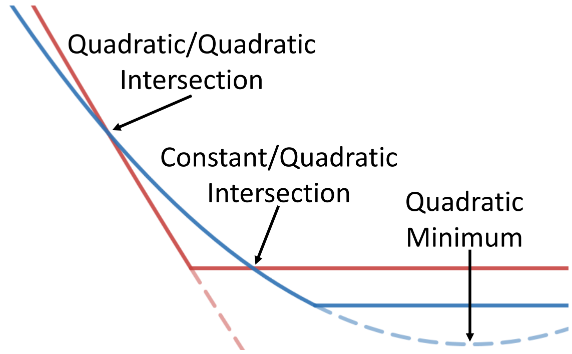


Figure 5.5: The set of possible elastic objective intersections.

The first is where the quadratic terms are equal:

$$A_{P_j}U^2 - B_{P_j}U = A_{P_k}U^2 - B_{P_k}U$$

If $A_{P_j} \neq A_{P_k}$ and $B_{P_j} \neq B_{P_k}$, then this is:[16]

$$U = \frac{B_{P_j} - B_{P_k}}{A_{P_j} - A_{P_k}} \tag{5.20}$$

This value is calculated, then checked to ensure it is within the considered region and does not exceed $U_{P_j}^{\max}$ and $U_{P_k}^{\max}$ (in which case it lies outside the quadratic objective component).

The other intersection point is where the larger of the two PHI objectives' constant components intersects the quadratic component of the other. This can only happen at a single point not exceeding the utilization for which the quadratic component is minimized, as shown in Figure 5.5. Without loss of generality, assume $O_{P_j}^*(U_{P_j}^{\max}) < O_{P_k}^*(U_{P_k}^{\max})$. Then the intersection occurs where:

$$A_{P_j}U^2 - B_{P_j}U = O_{P_k}^*(U_{P_k}^{\max})$$

Solving for the smaller value of U:

$$U = \frac{B_{P_j} - \sqrt{B_{P_j}^2 + 4A_{P_j}O_{P_k}^*(U_{P_k}^{\max})}}{2A_{P_j}} \tag{5.21}$$

[16]They also trivially intersect at $U = 0$, but this point is not considered.

Again, this value is checked to ensure it is both within the considered region, and within the range of utilizations for which $O^*_{P_j}$ is quadratic and $O^*_{P_k}$ is constant — or vice versa, if $O^*_{P_k}(U^{\max}_{P_k}) < O^*_{P_j}(U^{\max}_{P_j})$.

Once intersections have been identified, the region is split at those points, and each sub-region is associated with the PHI with the lower objective value according to Equation 5.17. Adjacent sub-regions with the same PHI are merged before replacing the region R_k.

Execution Time of Online Adjustment

The produced set of regions $\mathbf{R}$ is a sorted lookup table over the entire utilization range that can accommodate the task set. Each region R_k is associated with the PHI P_j that minimizes the elastic objective for the corresponding utilization interval. Thus, for a given utilization bound U_D, binary search finds the optimal PHI in polynomial time, as we now prove. The period $T^{U_D}_{1,j}$ can then be assigned to task τ_1 according to Equation 5.16, with other task periods assigned according to $T_i = a_{i,j} \cdot T^{U_D}_{1,j}$.

Theorem 9. *For a set of regions* $\mathbf{R}$ *constructed over* $|\mathbf{P}| = h$ *projected harmonic intervals,* $|\mathbf{R}|$ *does not exceed* h^2*.*

Proof. We prove this by induction. Clearly if $h = 1$, the number of elements of $\mathbf{R}$ is 1, which is h^2.

The algorithm generates regions by iterating over the set of PHIs. Assume that it is constructing a set of regions for $h+1$ PHIs. Further assume that after iterating over the first h PHIs, it holds that for the number x of elements in $\mathbf{R}$, $x \leq h^2$.

The insertion of PHI P_{h+1} can add a region to $\mathbf{R}$ for every point at which its objective $O^*_{P_{h+1}}$ intersects the objective $O^*_{P_i}$ for each PHI i, $1 \leq i \leq h$. As there are two such possible intersections for each PHI, this can produce up to $2h$ additional regions. Furthermore, an additional region might be added if $U^{\min}_{h+1}$ is less than $R^{\min}_1$, the current minimum utilization covered by $\mathbf{R}$. Alternatively, an additional region might be added if $U^{\min}_{h+1} > R^{\min}_j$ for the first region R_j where $U^{\min}_{h+1} < R^{\max}_j$. As both conditions cannot be true, it follows that only up to $2h+1$ additional regions may be added.

Then the total number of regions after insertion of P_{h+1} does not exceed

$$x + 2h + 1 \leq h^2 + 2h + 1 = (h+1)^2$$

$\square$

Corollary 3. *For n tasks and $k = T_n^{\max}/T_1^{\min}$, the number of regions in $\mathbf{R}$ does not exceed $k(n-1)^{2\lfloor \log k \rfloor}$.*

Proof. From Theorem 8, we know that for the number of PHIs h does not exceed $k(n-1)^{\lfloor \log k \rfloor}$. Then from Theorem 9, for h PHIs the number of regions does not exceed h^2. $\square$

Corollary 4. *For n tasks and $k = T_n^{\max}/T_1^{\min}$, binary search over the set of regions $\mathbf{R}$ takes time $\boldsymbol{\mathcal{O}(\log^2 k + \log k \cdot \log n)}$*

Proof. For a sorted list of x elements, binary search requires $\mathcal{O}(\log x)$ time. Then from Corollary 3, $|\mathbf{R}| \leq k(n-1)^{2\lfloor \log k \rfloor}$. So binary search proceeds in time:

$$\log\left(k(n-1)^{2\lfloor \log k \rfloor}\right) = 2\lfloor \log k \rfloor \left(\log k + \log(n-1)\right)$$

This is $\mathcal{O}(\log^2 k + \log k \cdot \log n)$, which is polynomial in the length of the input. $\square$

5.6 Implementation Considerations

In this section, we discuss several considerations for implementation in real systems, both for elastic scheduling in general, and specifically with harmonic period constraints.

5.6.1 Characterizing Elasticity

The elasticity constant E_i assigned to task τ_i is intended to represent "the flexibility of the task to vary its utilization" [39]. Under this interpretation, a natural assumption is that elasticities should be assigned according to the functional semantics of a task or application, independently of the platform it will run on. However, as we will show, an interpretation of elasticity that considers the first-order impacts on result utility (e.g., control performance) of adjusting task rates requires a dependence on execution times, which implies that elastic values must be reassigned depending on the target platform.

The elastic objective (Expression 5.1a) formulated by Chantem et al. in [44, 45] suggests that *loss* in result utility is impacted by compressing task utilizations. From this, the first-order error induced by individually decreasing the rate R_i of task τ_i from its fastest desired value $R_i^{\max}$ can be described as:

$$\mathcal{L} = w_i(R_i^{\max} - R_i)^2 \tag{5.22}$$

where $w_i = C_i^2/E_i$ and $\mathcal{L}$ is some measure of loss or error. Then for a given application, if the set of weights $\{w_i\}$ are obtained, each task's elasticity can be assigned as:

$$E_i = \frac{C_i^2}{w_i} \tag{5.23}$$

This implies that if result error is a function of task invocation rates, then elastic constants depend on the platform-specific execution times of each task. If elastic scheduling is used as a technique to adjust a task system for portability across lower powered platforms on which it might not otherwise be schedulable, the elastic constants must be reassigned for each target.

We use this interpretation in the next section to assign elasticity constants to each task in our evaluated applications.

5.6.2 Online Adjustment

As we have discussed in previous chapters, elastic scheduling is especially important in the context of *online* adjustments to task periods during admission control of new tasks, or when available computational resources change. In [39], Buttazzo et al. argue that period *increases* in response to overload may happen at any time, as existing jobs will not miss the new (extended) deadline. However, if resources become available, a task's period must wait to *decrease* until after its active job has completed; otherwise, the job might miss the new (shorter) deadline. During these brief intervals, the task system may execute at a lower utilization than can be accommodated, but this is not a problem for schedulability.

With the addition of harmonic period constraints, however, some task periods might *decrease* in response to reduced available utilization. We illustrate this with the following example.

Example 2. *Consider the set of tasks with the parameters in Table 5.1.*

	$T_i^{\min}$	$T_i^{\max}$	C_i	E_i
τ_1	5	6	0.3	3
τ_2	12	17	0.7	4
τ_3	23	36	0.1	1

Table 5.1: Elastic Tasks with Harmonic Period Constraints

If the available utilization U_D is 0.12*, the tasks will be assigned periods* $\{6, 12, 24\}$*. However, if available utilization drops to* 0.11*, task periods will be reassigned as* $\{5, 15, 30\}$*. In this case, the period of τ_1 decreased.*

This example illustrates that the policy suggested by Buttazzo et al. in [39, 40] to simply extend periods when new tasks arrive or available utilization decreases might not extend to elastic scheduling of harmonic task sets. If the new period is applied immediately, a running job might miss its deadline; but if the period change is delayed, the task system may remain overloaded. Moreover, delaying the period change would mean that for some time interval, periods are not harmonic, which may have functional implications in systems that require temporal alignment among tasks.

Evaluation of policies to address this challenge are outside the scope of this dissertation. Nonetheless, we propose three possible approaches:

- Require that changes in available utilization only occur at hyperperiod boundaries. For periodic task sets, the period adjustment is guaranteed to occur when there are no active jobs, avoiding issues with alignment, overload, and missed deadlines. With harmonic periods, the hyperperiod is equal to the maximum of the task periods, so this is not a highly restrictive policy. If the scheduler has full control over resource allocations, this may be feasible; however, in dynamic or mixed-criticality systems, scenarios might arise where the system cannot control the instant at which reallocations of processor utilization have to be made.

- In a mixed-criticality system, some degree of job dropping may be acceptable. In this case, a policy might allow tasks to restart, dropping jobs if necessary, to maintain alignment after a new set of periods are assigned.
- However, if job dropping remains unacceptable, an alternative approach would proportionally extend task periods for a single hyperperiod to prevent any from decreasing. In the case of Example 2, on reassignment to $\{5, 15, 30\}$, periods would instead be extended to $\{6, 18, 36\}$ for 36 time units — values are selected from the same projected harmonic interval, but using a base that guarantees that no period decreases. However, this might cause some task periods to extend above their maximum constrained value; indeed, in this case, T_2 becomes 18, whereas $T_2^{\max} = 17$. Nonetheless, this may be preferred to job dropping in scenarios where the transition cannot wait until the hyperperiod.

Further development and evaluation of these policies is left to future work.

5.7 Evaluation

In this section, we evaluate our model and algorithms in the context of two real-world applications: FIMS and ORB-SLAM3. These applications are composed of relatively small task sets, so we perform additional evaluations over larger sets of synthetically-generated tasks.

5.7.1 FIMS

The Fast Integrated Mobility Spectrometer (FIMS) [167] is a flown instrument that characterizes atmospheric aerosols. Recent efforts to enable real-time measurements of aerosol particle sizes (e.g., to instruct the aircraft to follow an aerosol plume) achieved the desired performance on a Raspberry Pi 4 [166]. The improved computational pipeline illustrated in Figure 5.6 captures and processes images to detect particles, grouping them into fixed-duration windows of particle inlet time. Using matrix inversion, it converts instrument responses from particle spatial coordinates to determine the particle size distribution within each window. “Housekeeping” (HK) data readouts from other sensors (e.g., temperature and pressure) are synchronized with the inversion process, and are used to determine if a new data inversion matrix must be computed. Harmonic execution guarantees a fixed, integral number of time windows associated with each job. This stabilizes the fraction of particles lost during the inversion process, ensuring a consistent representation of particle distribution and maintaining measurement accuracy, while also bounding the latency between a particle’s inlet time and its subsequent inclusion in the reported size distribution.

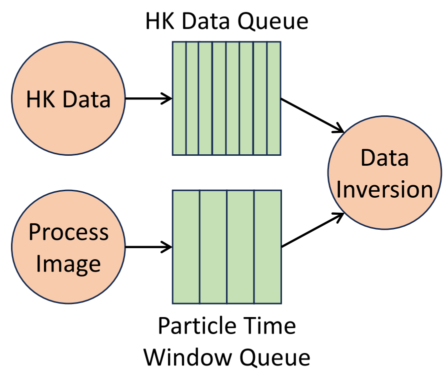


Figure 5.6: The FIMS Computational Pipeline

In our evaluation, we consider a future deployment where FIMS runs concurrently with other applications (e.g., flight control, SLAM, and telemetry) atop SWaP-constrained hardware, e.g., on a UAV. To examine its portability across embedded platforms, we experiment with constraining FIMS to just a fraction of the bandwidth of a *single* core of the Pi 4 and verify that, by using harmonic elastic scheduling to adjust the FIMS task periods, we can avoid missing deadlines.

Experimental Setup

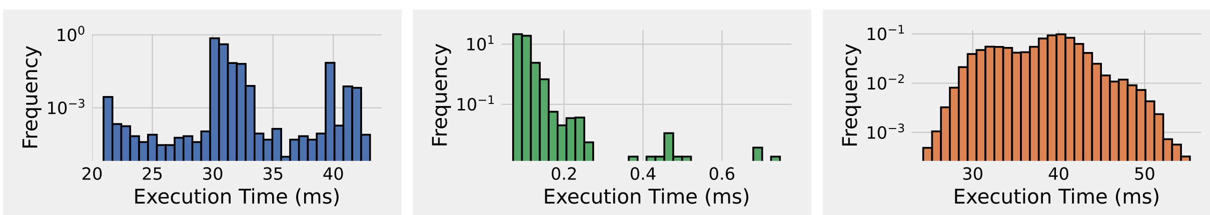
We run FIMS on a Raspberry Pi 4 Model B, which has a 4-core, 64-bit Cortex-A72 (ARM v8) CPU running at 1.50GHz and 4GB of RAM. We test with Linux 5.10.103 and disable CPU throttling. As in Section 2.4.1, we also disable real-time throttling by writing “-1” to the file `/proc/sys/kernel/sched_rt_runtime_us`, isolate CPU core 3 from the scheduler, and run FIMS on just that core. Image processing, HK data reading, and data inversion are all run under the `SCHED_FIFO` real-time scheduling class using rate-monotonic priorities — 98, 97, and 96 respectively — with priority 99 reserved for an *interference task* that we will describe later.

Lacking direct access to the prototype FIMS instrument, we instead use an offline dataset consisting of camera images from its particle chamber and readouts from its other sensors. To avoid delays from disk I/O, we load these into program memory prior to execution. In a complete implementation, separate threads will perform the memory transfer from attached USB devices asynchronously.

A complete run with our experimental dataset handles 12 000 images, processed at a period of 100 ms. HK data is read every 500 ms and data inversion runs every second.

Profiling Execution Times

To measure execution times associated with each task, we make calls to `getrusage()` when each job completes, measuring the total CPU time (user and system) consumed by the task since the end of the prior job. This accounts for execution of the task's function plus the overhead of context switching and timer handling. To capture worst-case conditional behavior, we force recalculation of the inversion matrix with each iteration of data inversion. Profiling results over 10 complete runs are plotted in Figure 5.7.

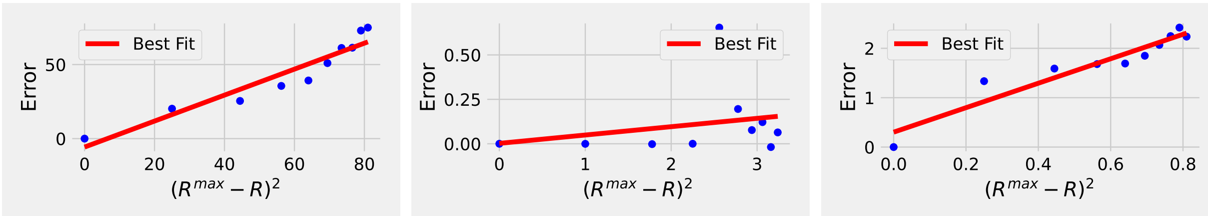


(a) Image (max 43.0ms). (b) HK (max 0.747ms). (c) Inversion (max 55.3ms).

Figure 5.7: FIMS task execution time distributions.

Assigning Elasticity Values

We assign elasticity values to tasks per the methodology in Section 5.6.1. We define loss as $1000 \cdot (1 - \theta)$, where θ is the cosine similarity between the distribution of particle sizes produced by a period-adjusted run of FIMS and the ground-truth values. We measure the loss associated with adjusting task periods individually. To reflect the real instrument's behavior, when increasing the image processing period we stack successive image frames. When increasing the HK data period by a factor of $n\times$, we sample every n^{th} data point, interpolating over the most recent two. For data inversion, we change the time bin over which particle size distributions are measured to remain equal to the period. Results are plotted in Figure 5.8.



(a) Process Image. (b) HK Data. (c) Data Inversion.

Figure 5.8: FIMS errors resulting from reduced task rates.

Using linear regression over the measured error values for each task τ_i, with $(R_i^{\max} - R_i)^2$ as the independent variable, we obtain values of w_i according to Equation 5.22, then use these to compute values of E_i per Equation 5.23. Table 5.2 summarizes the assigned parameters.

	$T_i^{\min}$	$T_i^{\max}$	C_i	E_i
Process Image	100	1000	43.0	2.11
HK Data	500	5000	0.747	0.012
Data Inversion	1000	10000	55.3	1.23

Table 5.2: FIMS Task Parameters

Evaluating Scalability

With these parameters, FIMS demands a maximum utilization of 0.487. To test under tighter constraints, we run a highest priority *interference task* that limits the CPU utilization available to FIMS: it registers an interval timer with a period of 50 ms and spins for a programmable length of time. We run FIMS concurrently with the interference task using different busy loop durations, adjusting the FIMS task periods according to our harmonic elastic model. We then measure each FIMS job's latency (elapsed wallclock time) from task release to completion; results are shown in Table 5.3.

U_D	T_{IMAGE}	T_{HK}	T_{INV}	$L^{\max}_{\text{IMAGE}}$	$L^{\max}_{\text{HK}}$	$L^{\max}_{\text{INV}}$
0.5	100	500	1000	67	305	305
0.4	115	575	2298	94	489	1275
0.3	147	881	9682	137	709	822
0.2	222	3325	9973	222	1324	1327
0.1	458	3205	9615	348	2784	2785

Table 5.3: FIMS elastic task period assignments and latencies when running concurrently with interference task.

$L^{\max}$ indicates the longest measured latency over 3 complete runs; this value never exceeds the corresponding task period, indicating that ***no deadlines were missed.***

5.7.2 ORB-SLAM3

ORB-SLAM3 [42] is a visual-inertial simultaneous localization and mapping (SLAM) system widely used in autonomous vehicle and robotics applications that supports stereo camera inputs. Its data-driven computational pipeline is illustrated in Figure 5.9. Object tracking fuses captured image frames with interstitial inertial measurements to detect feature points and generate descriptions. The resulting metadata-rich keyframes are matched against a map database of prior descriptions to determine the system's position. If the current environment differs sufficiently from the existing map, the map is updated within the backend mapping task. A loop closing task is activated aperiodically to identify potential trajectory loops as the vehicle moves, allowing for the calibration of the vehicle's position against the map database.

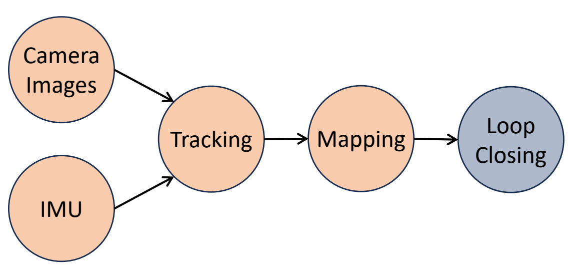


Figure 5.9: The ORB-SLAM3 Computational Pipeline

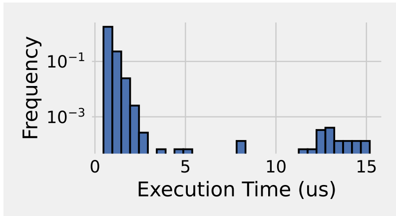
Our goal is to consider the portability of ORB-SLAM3 to SWaP-constrained hardware, where it may execute concurrently on a single core with other applications; the utilization U_D available to it may change during runtime. When computational resources are insufficient to guarantee completion of all tasks, frames may be dropped or desynchronized, giving increasingly erroneous results [90]. While missed deadlines do *not* necessarily result in system failure, we show that adjusting task periods in a principled way offers better localization than ORB-SLAM3's baseline implementation.

Experimental Setup

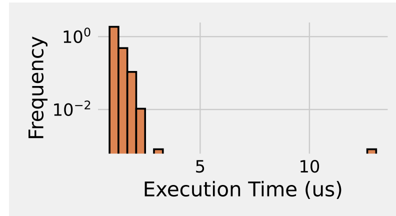
We run ORB-SLAM3 on a 6-core Intel i7-4960X running at 3.6GHz with 12GB of RAM using Linux 4.9.30 built with LITMUSˆRT [41]. We disable HyperThreading and CPU throttling. We evaluate in simulation using the EuRoC MAV dataset [33] from drones in real-world environments. Realistic timing is achieved by using ROS [125] to deliver image frames and inertial measurement unit (IMU) data as messages. IMU data arrives every 5 ms, and camera frames every 50 ms; a single trace includes 187 seconds worth of data. Tracking executes with as period of 50 ms, and by default, mapping executes only when tracking identifies a keyframe (with a minimum period of 50 ms).

Profiling Execution Times

We measure the execution times of each task by successive calls to `clock_gettime()` using `CLOCK_THREAD_CPUTIME_ID`. Results for each task, profiled over a complete trace, are shown in Figure 5.10.

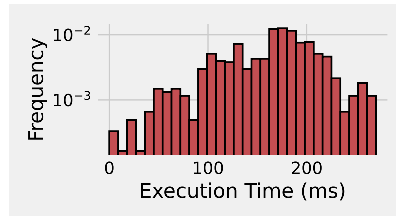


(a) IMU (max 15.2μs).



(b) Camera (max 13.1μs).

(c) Tracking (max 31.3ms).



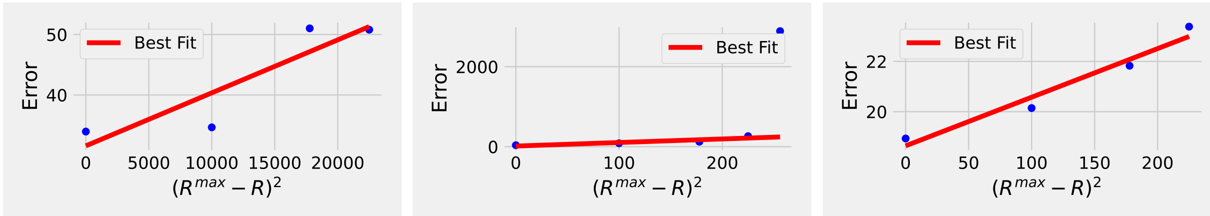
(d) Mapping (max 270ms).

Figure 5.10: ORB-SLAM3 task execution time distributions.

Assigning Elasticity Values

For ORB-SLAM3, we define loss as $10^4 X$, where X is the relative translational error (RTE) of the completed map in units of meters from the ground truth. We again capture the first-order effects of adjusting task periods individually. Here, we treat the two stereo image processing threads and the tracking thread as a single task (the application's dataflow pipeline requires these to run at the same rate). We simulate increases in the IMU and image task periods by dropping message frames (e.g., for an IMU period of 10 ms, we drop every other IMU message).

To achieve more deterministic execution time behavior, we modify tracking to categorize every image frame as a keyframe — in a non-degraded state, mapping will perform updates for every frame. However, if utilizations are compressed due to overload, keyframes are selected according to the harmonic relationship between the mapping and tracking periods. To characterize the impact on loss, we hold other periods constant while decreasing the number of frames selected as keyframes (thus increasing the mapping period). We slow the replay of the EuRoC dataset to allow mapping to process every frame as a keyframe without overrunning its deadline. Results are plotted in Figure 5.11.



(a) IMU. (b) Cam. + Track. + Map. (c) Mapping only.

Figure 5.11: ORB-SLAM3 errors resulting from reduced task rates.

We again use linear regression to fit values of w_i (Equation 5.22). Observe that in Figure 5.11b, the last point (corresponding to a period of 250 ms) diverges sharply from the linear trend. This suggests unstable behavior, so we limit the fit (and $T_i^{\max}$) to 200 ms. Note also that when increasing the image and tracking periods, the mapping period is also increased (its period cannot be *less* than these tasks); we adjust the first-order weight that characterizes the image and tracking task accordingly. After computing values of E_i (Equation 5.23), we obtain the task parameters listed in Table 5.4.

	$T_i^{\min}$	$T_i^{\max}$	C_i	E_i
IMU	5	20	0.015	0.263
Camera + Tracking	50	200	31.3	4006
Mapping	50	1200	270	1.14e5

Table 5.4: ORB-SLAM3 Task Parameters

Evaluation of Online Adjustment

We artificially limit the CPU utilization available to ORB-SLAM3 by pinning all of its threads on a single core and restrict its total bandwidth with the Linux `cpu` control group (cgroup). We modify the beginning of its main loop to:

1. Select a random amount of available utilization between 0.50–0.75.
2. Enforce this by updating the cgroup.
3. Run the online algorithm described in Section 5.5 to search the lookup table (generated during program initialization) for the best PHI corresponding to that utilization.
4. Update task periods accordingly.

This is invoked every second to allow completion of at least one hyperperiod, and incurs an overhead of <22 μs to select and update the period assignments.

We run ORB-SLAM3 in this manner over EuRoC trace `MH_01`, comparing its accuracy to an unmodified (non-adaptive) baseline version of the program. Results are plotted in Figure 5.12.

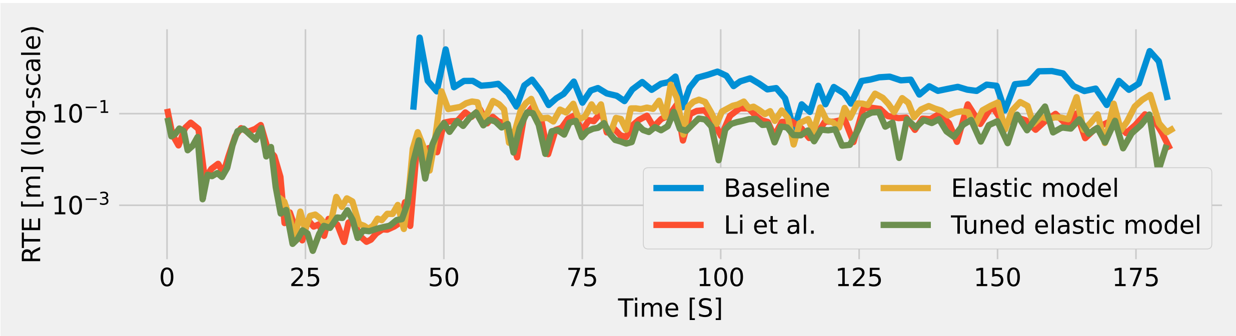


Figure 5.12: Comparison of RTE for different adaptive variants of ORB-SLAM3.

In our constrained execution environment, the mean RTE of the baseline implementation (Baseline) is 89 mm, and only succeeds in mapping the final 139s of the flight. In contrast, the adaptive implementation (Elastic model) achieves a mean RTE of 13 mm, a 6.6× reduction, and maps the final 164s. Using insights from past experience, we also try using the mean observed execution times for the mapping, camera, and tracking workloads (Tuned elastic model). This achieves even better accuracy: the mean RTE is only 8.6 mm, a 10.4× reduction, and maps the entire flight.

Finally, we compare our results to the adaptive approach of Li et al. in [90] where ORB-SLAM3 is integrated with an online machine learning model that predicts budget constraints and adapts execution via selective frame dropping, achieving a mean RTE of 9.3 mm. Not only does ***our elastic model achieve better localization***, it is more suited for broad use across real-time applications than the approach in [90]: on a hard real-time system, task dropping may be unacceptable, so our elastic model extends task periods to avoid missed deadlines. Furthermore, our approach enables straightforward adaptation of execution parameters using only a basic characterization of the impacts on control performance and does not require highly-tailored ML-based integration with the existing application.

5.7.3 Evaluation with Larger Synthetic Task Sets

The ORB-SLAM3 and FIMS task sets are relatively small. In this subsection, we explore how task set size impacts the performance of the algorithms discussed in Section 5.4.2 and Section 5.5 using larger task sets with randomly-generated parameters.

Experimental Setup

We generate 1000 task sets each of sizes 5–50. The minimum period $T_i^{\min}$ of each task τ_i is sampled from the log-uniform distribution described in [61] over integers in the range $[1, 100]$. A maximum period $T_i^{\max}$ is obtained by multiplying each $T_i^{\min}$ by a value selected uniformly in $[1, 10]$; this bounds the ratio k of the largest to smallest periods to $k \leq 1000$. Each task set is assigned a total maximum utilization of 1;[17] individual task utilizations $U_i^{\max}$ are then assigned using the UUniSort algorithm [24]. C_i is derived as $U_i^{\max} \cdot T_i^{\min}$. Weights w_i are selected uniformly in $[0, 1]$, from which elasticities E_i are computed according to Equation 5.23. Tasks are then sorted by $T_i^{\min}$. We implement the algorithms in C++, and compile them with GCC optimization level `-O3`. All experiments are run on a single core of an AMD EPYC 9754 with 128GB of RAM running Linux 5.14.0.

The Harmonic Period Problem

We first evaluate the proposed approach to the harmonic period problem described in Section 5.4.2. For each number of tasks, we count how many of the 1000 corresponding sets had a feasible harmonic period assignment. Results are illustrated in Figure 5.13, which shows that as the number of tasks increases, the proportion of task sets for which a solution exists goes down.

For those same task sets, we also measure how long it took to find a feasible harmonic period assignment — or to determine that such an assignment does not exist — and for those where an assignment *does* exist, we count the number of harmonic zones projected onto the last

[17] Generality is not lost, as we are considering the space of parameters as it impacts algorithmic performance, not to test schedulability ratios.

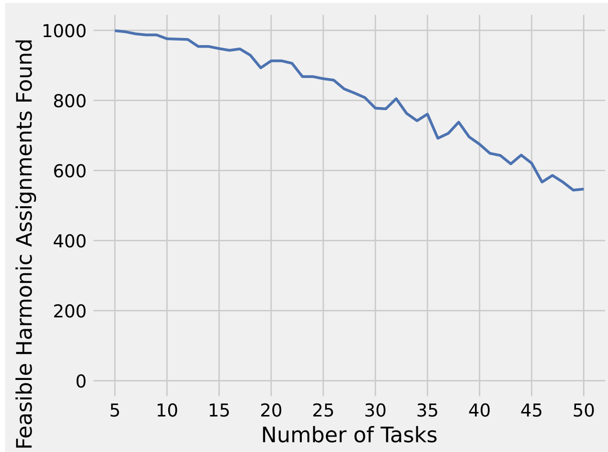


Figure 5.13: Harmonic assignments found.

interval. Figure 5.14 plots the maximum value of each metric observed for each number of tasks.

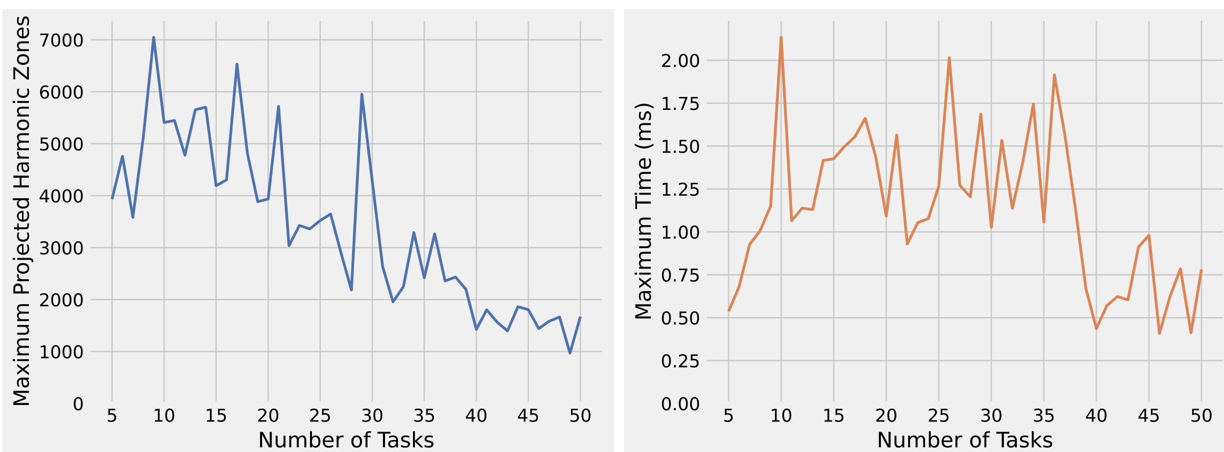


Figure 5.14: Harmonic assignments found.

We observe that the maximum number of harmonic zones projected onto the last interval also *decreases* as the number of tasks increases, even though the theoretical bound increases. We conjecture that this is because adding more period intervals will typically impose tighter constraints on the harmonic search. We also see that ***the algorithm is efficient***, executing in under 2.2 ms, and that its measured maximum execution times do not have a tight relationship with the number of harmonic zones projected onto the last interval.

The Ordered Harmonic Elastic Problem

We next evaluate both the naïve and efficient approaches to the ordered harmonic elastic problem described in Section 5.5. For those task sets where a harmonic assignment is possible, we count how many projected harmonic intervals (PHIs) are found, how long it takes to generate the lookup table (LUT), and measure the time it takes to iterate over all PHIs to find the optimal period assignment. Results for each number of tasks are illustrated in Figure 5.15.

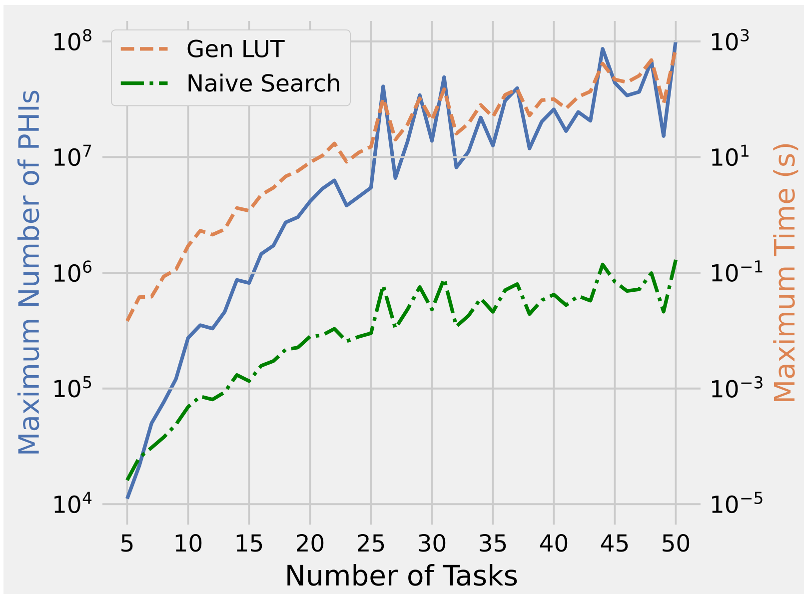


Figure 5.15: Comparison of the maximum number of PHIs with the maximum time to iterate over them to find the optimal PHI for a given utilization bound and to construct the LUT.

As expected, we observe that the maximum number of PHIs grows rapidly with the number of tasks (note the logarithmic scale of the y-axes). The time to generate the lookup table, as well as for the naïve search, are roughly proportional to the number of PHIs. ***The naïve search does not exceed 170 ms***; it is reasonably efficient, and provides greater flexibility as it allows for online admission of new harmonic tasks, though it might be too slow for online use with larger task systems. Times to generate the LUT remain under 10 ms for up to 8 tasks, but reach as high as 12.6 minutes for up to 50 tasks. This suggests that for smaller task systems, recomputing a set of harmonic assignments *online* if a new elastic task is admitted to the system may be feasible.

We also measure the maximum time it takes perform a binary search over the LUT once it has been generated. These results are plotted side-by-side with the maximum LUT size for each number of tasks in Figure 5.16.

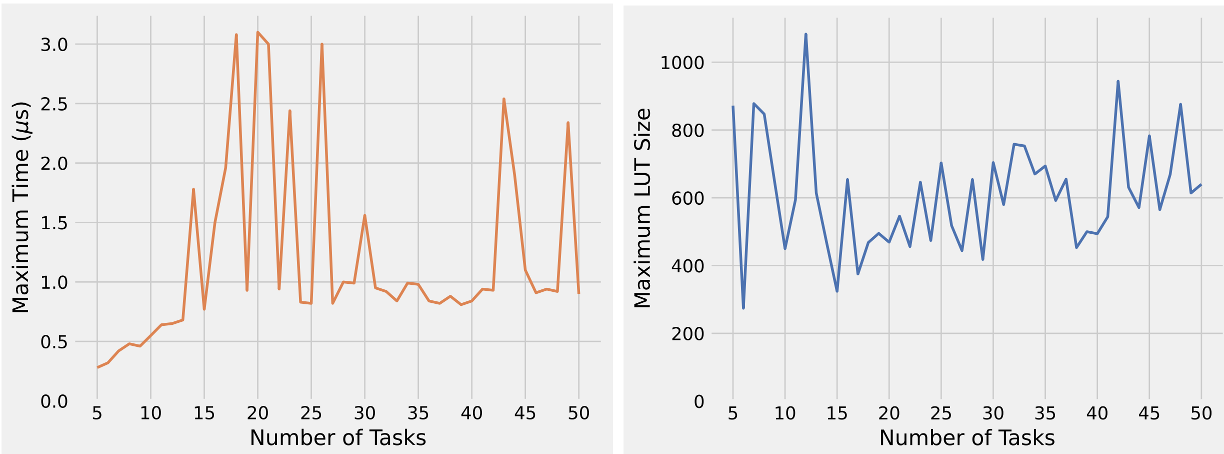


Figure 5.16: Maximum LUT sizes and times to perform binary search for each number of tasks.

We observe that for larger task sets, slow offline computation of the lookup table nonetheless ***allows for rapid online adaptation of existing task periods***: searching the LUT and assigning new periods does not exceed 3.2 μs. Furthermore, ***the LUT remains small*** — it did not exceed 1083 entries for those task sets that we tested — making it feasible for deployment even on memory-constrained embedded systems.

5.8 Conclusions

In this chapter, we have considered how to extend elastic scheduling to systems of tasks with harmonic periods. We argue that, if every task's period is constrained to a continuous interval, then finding a set of harmonic periods is unlikely to have a polynomial time solution. However, ***we outline a pseudo-polynomial algorithm to solve the problem***, and show it is efficient even for large task sets.

We then turn our attention to the problem of finding an *optimal* assignment of harmonic periods under the elastic model. If a total order is imposed on task periods, the problem can be solved efficiently for a small number of tasks. Alternatively, ***an offline lookup table can be constructed that allows rapid reassignment of task periods if available utilization changes***.

We demonstrate that our approach allows applications such as FIMS to be deployed effectively atop SWaP-constrained hardware platforms, and that it allows ORB-SLAM3 to adjust its task periods in response to runtime interference.

In our evaluation, we identified the formal connection implied by the elastic objective between elasticity constants and control performance. Quantifying the impact on result error of increasing individual task periods lets us successfully assign values to each task, and for ORB-SLAM3, this approach allowed for a 10.4× decrease in localization error compared to a baseline implementation on a dynamic resource-constrained system.

5.9 Acknowledgements

I would like to thank Daisy Wang of Washington University in St. Louis. This chapter used Daisy's implementation of the real-time FIMS pipeline [166], and she collected all execution times and errors illustrated in Figures 5.7 and 5.8, which were necessary to assign the task parameters in Table 5.2. My model produced the period assignments in Table 5.3 for each utilization bound, which Daisy then applied to the FIMS pipeline to produce the latency results in that same table.

I would also like to thank Ao Li of Washington University in St. Louis, who was instrumental in linking my implementation of the harmonic elastic algorithms into the ORB-SLAM3 application, and writing the test harness to change its available utilization with Linux control groups and apply the periods produced by the model to the SLAM pipeline tasks. Ao also showed me how to run each version of his pipeline, and produced several of the traces used to produce the execution time and error plots in Figures 5.10 and 5.11.

Chapter 6

Subtask-Level Workload Compression for Parallel DAG Tasks

6.1 Introduction

To this point, this dissertation has considered elastic scheduling of *sequential tasks* on a uniprocessor, and — to a lesser extent — on multiple processors. However, the growing prevalence of multicore CPUs, even in embedded platforms, has enabled increasingly complex real-time applications to exploit *intra-task parallelism*. Systems of tasks that individually require parallel execution on more than one processor to meet their deadlines are found in autonomous vehicles [82], computer vision systems [60], mobile robotics [4], hybrid structural and earthquake simulation [63, 64], and our own work on satellite telescopes [144, 171, 75]. This has inspired extensions of the elastic framework to *federated scheduling* [94] of parallel real-time tasks, under which each high-utilization parallel task[18] is allocated dedicated processor cores in sufficient number to guarantee schedulability. This class of models will be the focus of the remainder of this dissertation.

In the prior work of Orr et al. [119], if the total number of allocated cores exceeds the number available in the system, parallel task utilizations are compressed by decreasing their workloads over a continuous range until the demand for processors can be met. Utilizations thus assigned satisfy a reformulation of the quadratic optimization problem presented by Chantem et al. in [44, 45] that is solved by Buttazzo's original elastic scheduling model [39, 40].

[18]Tasks with utilization $U > 1$ that must execute in parallel on more than one core are high-utilization. Tasks with utilization $U \leq 1$ are low-utilization, and individually require only a single core.

6.1.1 Limitations of the Prior Work

The approach of Orr et al. in [119] has three fundamental limitations.

- First, it decreases the total computational workload of the task as a whole, **without consideration of the impact on each individual *subtask*.** The ability of each subtask to vary its workload — and the resulting impact on quality of outcome (e.g., control performance, prediction accuracy, etc.) — should be considered individually [100, 101, 140, 7, 146].

- Second, it allocates processor cores according to the methodology in [94], which considers each parallel task's total workload, deadline, and *span*. As it decreases task workloads, the model in [119] holds the span constant. However, **span may also decrease as individual *subtask* workloads are compressed**, allowing the system to reach a schedulable configuration with less overall compression. For this effect to be captured, an elastic model must be cognizant of the DAG structure induced by the precedence constraints among the subtasks composing each parallel task.

- Third, it only considers core allocation to high-utilization parallel tasks. In fact, under the federated scheduling model in [94], ***low-utilization* tasks are scheduled concurrently on any remaining cores not allocated to the high-utilization tasks**. The model of Orr et al. in [119] compresses parallel tasks given a number of available cores, only suggesting as an aside that low-utilization tasks can be compressed if there are cores remaining. However, jointly compressing *all* tasks in the system may change the number of cores separately allocated to high- and low-utilization tasks, and this effect should be captured by the elastic model.

6.1.2 Contributions of This Chapter

To address these limitations, we propose a new model of ***subtask-level elasticity for federated scheduling of parallel tasks*** in which each subtask is assigned a continuous range of acceptable workloads and its own elastic constant. This gives rise to an alternative expression of the quadratic optimization problem in [119] where the objective is to minimize the total compression applied to each subtask workload, while guaranteeing schedulability.

In this chapter, we propose to solve the problem by formulating it as a mixed integer quadratic program (MIQP). Constructing the MIQP is, for the most part, straightforward. The primary challenge arises from the representation of the task span. Because cores are allocated according to the workload, span, and deadline parameters of each task in [94], a span variable is introduced during the MIQP's construction. This chapter proposes two alternative methods to enforcing this intended representation of that variable as the sum of the assigned workloads along the critical path of the task DAG, then evaluates the efficiency with which the Gurobi Optimizer [72], a commercial off-the-shelf constraint solver, is able to solve MIQPs of each form.

In Chapter 4 of this dissertation, we observed that solving a single MIQP for a complete set of constrained-deadline sequential tasks may be inefficient; Section 4.7 proposed instead to construct an MIQP for each task individually, then demonstrated how the solutions for each MIQP may be used to solve the problem jointly for the complete task system. Applying this same insight to subtask-level elastic scheduling for parallel DAG tasks, we demonstrate an alternative approach whereby an MIQP is solved individually for each *task* for every feasible assignment of cores to that task. From the resulting collection of discrete compressed states and their objective function values corresponding to each core allocation for each task, a dynamic-programming (DP) algorithm can select a state for each task such that the joint assignment optimizes the quadratic objective.

We demonstrate that, despite the larger number of invocations of the solver, this dynamic-programming based approach is often more efficient than solving a single MIQP jointly over the collection of tasks. Moreover, ***it enables pseudo-polynomial task compression during dynamic runtime changes***, such as admission of new tasks, if those discrete states are determined offline for each task during characterization of their other parameters (e.g., control flow DAG, subtask execution times, etc.). Furthermore, it solves the problem of jointly allocating cores to low-utilization tasks, as each feasible core assignment to the overall collection of low-utilization tasks can itself be represented as a discrete state and incorporated into the DP problem.

6.1.3 Organization

The remainder of this chapter is organized as follows:

- Section 6.2 provides background on federated scheduling of parallel tasks and the existing elastic scheduling models used in that context.
- Section 6.3 highlights the limitations of those elastic scheduling models and motivates our new model of subtask-level elastic scheduling. It also provides the formal problem statement considered in this chapter.
- Section 6.4 shows how to construct an MIQP that can be solved with Gurobi to assign subtask workloads that satisfy the model. It proposes, analyzes, and compares two alternative representations of the task's span.
- Section 6.5 proposes a DP-based approach that allows MIQPs to be solved for each individual task, then an optimal joint assignment of subtask workloads to be obtained over all tasks. It also demonstrates that this approach also accommodates low-utilization elastic tasks scheduled alongside the high-utilization parallel tasks.
- Section 6.6 evaluates the proposed algorithms with large sets of synthetic task systems and compares them to the prior algorithm in [119] where workloads are compressed while span is held constant.
- Section 6.7 concludes the chapter.

6.2 Background

6.2.1 Uniprocessor, Implicit-Deadline Elastic Scheduling

Necessary background on Buttazzo's elastic scheduling model for recurrent, implicit-deadline tasks [39, 40] can be found in Section 2.2.2. In this section, we review just those background concepts necessary for the development of our subtask-level elastic scheduling model.

Buttazzo's elastic recurrent real-time workload model [39, 40] provides a framework for managing overload by reducing the utilizations of individual tasks until the total utilization no longer exceeds the schedulable bound. Recall that Buttazzo's approach is to compress each task's utilization such that it is reduced from its desired maximum *proportionally* to the task's elasticity parameter, subject to the constraint that it remains no less than the specified minimum.

In Buttazzo's original model [39, 40], compression is realized by adjusting each task's period T_i according to its new utilization, i.e., $T_i = C_i/U_i$. In [119], Orr et al. observed that a task can either be *rate-elastic* or *computationally-elastic*; for the latter, the task's period T_i remains fixed, and its execution time is updated according to $C_i = U_i/T_i$.

Chantem et al. [44, 45] demonstrated that the utilizations assigned under Buttazzo's model satisfy the following quadratic optimization problem, restated from Chapter 4.2.1:

$$\min_{U_i} \quad \sum_{i=1}^{n} \frac{1}{E_i}(U_i^{\max} - U_i)^2 \tag{6.1a}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} U_i \leq U_D \tag{6.1b}$$

$$\forall_i, \quad U_i^{\min} \leq U_i \leq U_i^{\max} \tag{6.1c}$$

This has allowed extensions of the elastic framework to other task models with schedulability tests that do not rely strictly on a utilization bound, including to *federated scheduling* of parallel real-time tasks [94] for which *periods* [120] or *workloads* [119] are adjusted in response to reduced utilization assignments.

6.2.2 Elastic Frameworks for Federated Scheduling

The federated scheduling model of Li et al. [94] deals with systems of independent, sporadic, parallel, implicit-deadline real-time tasks. Each task τ_i is characterized as in Section 2.2.1 with parameters for its workload C_i, period T_i, and deadline D_i. Since tasks are assumed to have implicit deadlines, $D_i = T_i$.

Each parallel task τ_i consists of a set of subtasks $\tau_{i,j}$ with a precedence relation $\prec$ over them. Each individual subtask $\tau_{i,j}$ is characterized by a workload $c_{i,j}$, representing its worst-case execution time. An individual subtask must execute sequentially — i.e., it is characterized by a sequence of instructions that must be completed in order, and take up to $c_{i,j}$ time to complete. The task workload C_i is expressed as the total $C_i = \sum_j c_{i,j}$ over the subtask workloads.

We assume that subtask execution is reëntrent; execution may be preempted by another subtask, and it need not resume executing on the same core in the system. Subtasks may run in parallel, except as constrained by the precedence relation $\prec$: if $\tau_{i,a} \prec \tau_{i,b}$, then $\tau_{i,a}$ must fully complete its execution before $\tau_{i,b}$ is scheduled. We say that subtask $\tau_{i,k}$ becomes *available* when all tasks $\tau_{i,j}$ for which $\tau_{i,j} \prec \tau_{i,j}$ have completed execution.

The partial-ordering of precedence over subtask execution that describes task execution gives rise to a standard *directed acyclic graph (DAG)* representation with a collection of vertices $v_{i,j}$ corresponding to subtasks $\tau_{i,j}$. A directed edge from vertex $v_{i,a}$ to $v_{i,b}$ exists if and only if $\tau_{i,a} \prec \tau_{i,b}$ and there is no $\tau_{i,c}$ for which $\tau_{i,a} \prec \tau_{i,c} \prec \tau_{i,b}$, i.e., $\tau_{i,b}$ directly succeeds $\tau_{i,a}$.

Each vertex of the DAG is assigned a weight $w_{i,j}$ equal to the workload $c_{i,j}$ of its corresponding subtask. The task workload C_i can therefore be thought of as the volume of the DAG. The DAG representation also gives rise to a parameter L_i representing the task's *span*, which is the weighted length of the DAG's critical path. One may think of the span as the earliest completion time of a job in the task, relative to its activation time, if given an infinite number of cores on which to execute. It is clear that for a task to be schedulable, $L_i \leq D_i$. These concepts are illustrated in the following example.

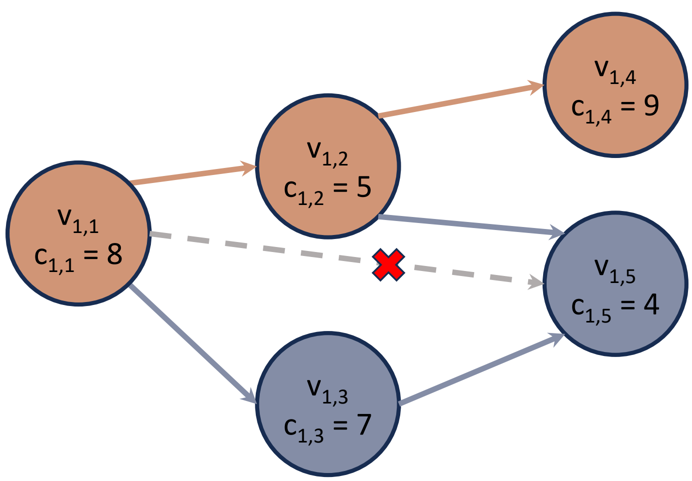


Figure 6.1: The DAG representation of the parallel task τ_1 in Example 3.

Example 3. *Consider a task τ_1 consisting of of five subtasks; $\tau_{1,1}$ with workload $c_{1,1} = 8$, $\tau_{1,2}$ with workload $c_{1,2} = 5$, $\tau_{1,3}$ with workload $c_{1,3} = 7$, $\tau_{1,4}$ with workload $c_{1,4} = 9$, and $\tau_{1,5}$ with workload $c_{1,5} = 4$. The total workload of task τ_1 is therefore $C_1 = 33$. A precedence relation over subtasks implies that $\tau_{1,1}$ must complete before $\tau_{1,2}$ and $\tau_{1,3}$ can begin execution, $\tau_{1,4}$ cannot begin until $\tau_{1,2}$ completes, and both $\tau_{1,2}$ and $\tau_{1,3}$ must complete before $\tau_{1,5}$ can begin.*

These relationships give rise to the DAG representation in Figure 6.1. Because precedence is a partial order relation, $\tau_{1,1}$ precedes $\tau_{1,5}$ by transitivity; however, since it does not directly precede $\tau_{1,5}$, there is no edge from $\tau_{1,1}$ to $\tau_{1,5}$.

The task's span L_1 can be computed as the length of the DAG's critical path. In this example, the critical path follows $v_{1,1}$, $v_{1,2}$, and $v_{1,4}$, as illustrated in Figure 6.1. The span is the total workload of the subtasks corresponding to vertices along the critical path; we calculate this as $L_1 = 22$.

Under Li et al.'s federated scheduling [94], each high-utilization parallel task τ_i is allocated m_i dedicated processor cores satisfying:

$$m_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil \tag{6.2}$$

In [120], Orr et al. extended the elastic framework to the federated scheduling model. If the total processor cores allocated exceed the number available, each high-utilization parallel task has its utilization compressed until the demand is met. Rather than a simple utilization bound, Expression 6.2 implies the following condition for schedulability:

$$\sum_{i=1}^{n} \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil \leq m \tag{6.3}$$

where m is the total number of processor cores available for high-utilization parallel tasks.

To find utilizations that satisfy Buttazzo's conditions for elastic scheduling, Orr et al. introduced the term λ representing the degree by which compression is applied to the task system. Recall from Equation 3.1 that it follows from the relationship in Equation 2.2 that for some value of λ, utilizations are assigned as:

$$U_i(\lambda) = \max\left(U_i^{\max} - \lambda E_i, U_i^{\min}\right) \tag{6.4}$$

The goal, then, is to find the *minimum* value of λ for which the task system becomes schedulable (i.e., for which Expression 6.3 is satisfied). This can be found to an arbitrary degree of precision via binary search over the range $[0, \lambda^{\max}]$. *However*, Orr et al. argue in [120] that this may result in wasted capacity due to the ceiling operator ($\lceil \cdot \rceil$) in Equation 6.2: if for some task τ_i the expression $(C_i - L_i)/(D_i - L_i)$ due to an assigned utilization $U_i(\lambda)$ is not an integer, then the period T_i can be reduced to some extent without affecting schedulability.

To avoid under-utilization, in [120] Orr et al. instead propose to assign utilizations according to the quadratic optimization problem in Expression 6.1, with the original schedulability condition (Expression 6.1b) replaced by Expression 6.3. In [119], Orr et al. extended their approach to *computationally-elastic* tasks, allowing parallel workloads to be adjusted over a continuous range: a task with period T_i would have its workload assigned as $C_i = T_i \cdot U_i$. This may be realized, for example, by reducing the quantity of input data to process or by forcing an iterative anytime algorithm to terminate early [146]. The span L_i is held constant.

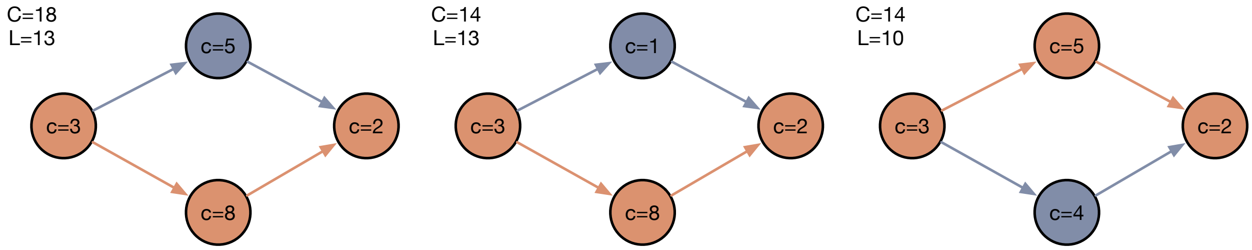
In this chapter, we address limitations of the model in [119] for federated scheduling of computationally-elastic tasks. In particular, we consider the individual implications of reducing the workloads of each *subtask*, both in terms of the relative flexibility or importance of each subtask on quality of outcome, as well as the effect on the task's span L_i. We also consider the implications of jointly compressing low-utilization tasks. The next section details these limitations and motivates our work.

6.3 Motivation and Limitations of Prior Work

6.3.1 Motivating a New Model of Subtask-Level Elasticity

The model in [119] for federated scheduling of computationally-elastic tasks holds the span L_i of each task τ_i constant while compressing workloads C_i. Depending on *how* the new workload assignment C_i is to be realized, i.e., which *subtask* workloads $c_{i,j}$ are to be reduced, the value L_i may also decrease, as Figure 6.2 illustrates. Without accounting for this, the model may be pessimistic in resource allocation and may over-compress task workloads.

Example 4. *Consider a task with parameters $C_i^{\max} = 10$, $L_i = 4$, and $D_i = 6$ to be scheduled on only 2 processor cores. If L_i is held constant, the task's workload would have to be decreased to $C_i = 8$ to satisfy Equation 6.2. But the workload needs only to be reduced by 1 unit along its critical path ($C_i = 9$ and $L_i = 3$) to be schedulable.*



(a) Uncompressed workload. (b) No change to span. (c) Span compressed.

Figure 6.2: Critical path may change depending on which subtask workloads are compressed.

Furthermore, the workload assigned to each individual subtask may uniquely impact result quality. Examples may be found in several application domains.

- **Distributed Systems.** The common end-to-end task model represents each task as a chain of subtasks distributed over multiple processors. To provide end-to-end QoS guarantees in open and unpredictable environments, EUCON [100, 101] provides a model predictive control approach to enforce CPU utilization bounds on distributed processors using a feedback loop. Individual processors, on which separate subtasks within a task's chain execute, are assigned different weights according to the relative importance of the corresponding subtask to the QoS demands or control performance of the system.

- **Autonomous Vehicles.** AutoE2E [8, 7] adjusts end-to-end task execution in autonomous vehicles to maintain schedulability in open and unpredictable environments. It has an inner rate-based control loop and an outer loop that adjusts the precision (execution time) of task workloads. It considers the relative importance of each subtask, both from the perspective of driver preference and control outcome. It also accomodates a nonlinear relationship between computational precision and execution time.

- **LiDAR Object Detection.** In [140], the authors explore fine-grained time and accuracy tradeoffs in the PointPillars [87] LiDAR object detector to enable adaptive execution in response to dynamic deadlines in an open environment. They represent the encoder pipeline as a parallel DAG, then analyze the execution time and corresponding accuracy associated with different levels of computational precision in various subtasks.

- **Prompt GRB Localization.** The Advanced Particle-astrophysics Telescope [30] (APT) is a planned satellite telescope that will detect and localize gamma-ray bursts (GRB) in real-time. In [146] and in Chapter 7, we model the localization pipeline as a highly-parallel fork-join task, then characterize the execution time and accuracy of the pipeline as functions of several computational input parameters corresponding to its various subtasks. This enables the pipeline to adapt its execution in response to dynamic workloads and deadlines due to GRB variability, selecting Pareto-optimal parameter values at job release to maximize accuracy while guaranteeing deadlines are met.

Each of these applications enables adaptive real-time execution based on the importance of each subtask. However, there is as yet no model that extends elastic scheduling to consider individual subtasks for parallel DAG tasks in general.

6.3.2 The Subtask-Level Elastic Workload Model

To fill this gap, we modify the model in [119] by assigning to *each* subtask $\tau_{i,j}$ a continuous range of execution times $[c_{i,j}^{\min}, c_{i,j}^{\max}]$ and an elasticity $E_{i,j}$. Under this new model, subtask workloads $c_{i,j}$ are selected to minimize a modified version of the objective in Expression 6.1a that considers the deviations of individual *subtask* utilizations from their desired values:

$$\min_{c_{i,j}} \quad \sum_{\tau_{i,j}} \frac{1}{E_{i,j} T_i^2} \left(c_{i,j}^{\max} - c_{i,j} \right)^2 \tag{6.5a}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} \left\lceil \frac{C_i - L_i(\{c_{i,j}\})}{T_i - L_i(\{c_{i,j}\})} \right\rceil \leq m \tag{6.5b}$$

$$\forall_{i,j}, \quad c_{i,j}^{\min} \leq c_{i,j} \leq c_{i,j}^{\max} \tag{6.5c}$$

We note that in prior work, Orr et al. [121] argue for a discrete model of computational elasticity, motivated by the fact that many applications have discrete modes or levels of precision that may be selected for computation. Nonetheless, we argue that a continuously-elastic workload model remains realistic in many systems. For example, at fine enough granularity, many discrete modes can be approximated as a continuous state space (e.g., the proportion of input data selected from a large set for processing). As another example, for workloads with anytime semantics — those that can terminate early and still provide a result — the number of iterations might be considered discretely. However, if execution times for each iteration follow some stochastic distribution, execution times may be selected from a continuous range to increase the likelihood of obtaining a better result. We discuss these scenarios in more detail in Chapter 7.

In the following sections, we demonstrate approaches to assigning workloads to subtasks under this model by using constraint programming solvers.

6.3.3 Joint Compression of Low-Utilization Tasks

Neither model for elastic scheduling of parallel tasks in [120, 119] address compression of low-utilization tasks. Both papers assume a fixed allocation of processor cores to high-utilization parallel tasks, and that Buttazzo's original techniques in [39, 40] can be applied to compress low-utilization tasks to be schedulable on the remaining processors. However, the semantics of elastic scheduling suggest that, as the allocation of cores to each task may change, so too might the allocation of cores among high- and low-utilization tasks.

Example 5. *Consider a system with $m = 4$ processor cores on which we must schedule the following 4 implicit-deadline tasks:*

1. $\tau_1 = (C_1 = 5, T_1 = 10)$, *a sequential task.*
2. $\tau_2 = (C_2 = 3, T_2 = 8)$, *a sequential task.*
3. $\tau_3 = (C_3 = 4, T_3 = 7)$, *a sequential task.*
4. $\tau_4 = (C_4 = 30, L_4 = 10, T_4 = 15)$, *a parallel task.*

The total utilization of the sequential tasks is ~ 1.45, and they therefore require 2 cores. However, τ_4 requires

$$\left\lceil \frac{30 - 10}{15 - 10} \right\rceil = 4$$

cores according to Equation 6.2. If we can compress the workload of τ_4 *along its span by* 5 *units, it is schedulable on the remaining 2 cores:*

$$\left\lceil \frac{25-5}{15-5} \right\rceil = 2$$

However, if we compress the utilizations of the sequential tasks so that they occupy a single core, then the workload (and span) of τ_4 *is only reduced by* $5/3$.

In Section 6.5 of this chapter, we present methods for joint compression of both high- and low-utilization tasks that allow dynamic allocation of processor cores to either set.

6.4 An MIQP for Subtask-Level Elastic Scheduling

The optimization problem in Expression 6.5 is naturally expressed as a mixed-integer quadratic program (MIQP), allowing it to be solved using one of many available off-the-shelf solvers.

6.4.1 Constructing the MIQP

We demonstrate an approach to constructing the problem for Gurobi [72], a mathematical optimization tool that solves MIQPs. Gurobi supports both integer and continuous variables, and both linear and quadratic expressions may be used as constraints or objectives. While we have chosen to focus on Gurobi, this approach should generalize to other quadratic solvers, including SCIP [2], which was used in Chapter 4.

As a running example, we use the task system Γ illustrated in Figure 6.3. Γ consists of parallel tasks τ_1 and τ_2. Task τ_1 has a total uncompressed workload $C_1^{\max} = 18$, corresponding span $L_1^{\max} = 13$, and a deadline equal to its period $D_1 = T_1 = 15$. Task τ_2 has a total uncompressed workload $C_2^{\max} = 23$, corresponding span $L_2^{\max} = 16$, and a deadline equal to its period $D_2 = T_2 = 19$.

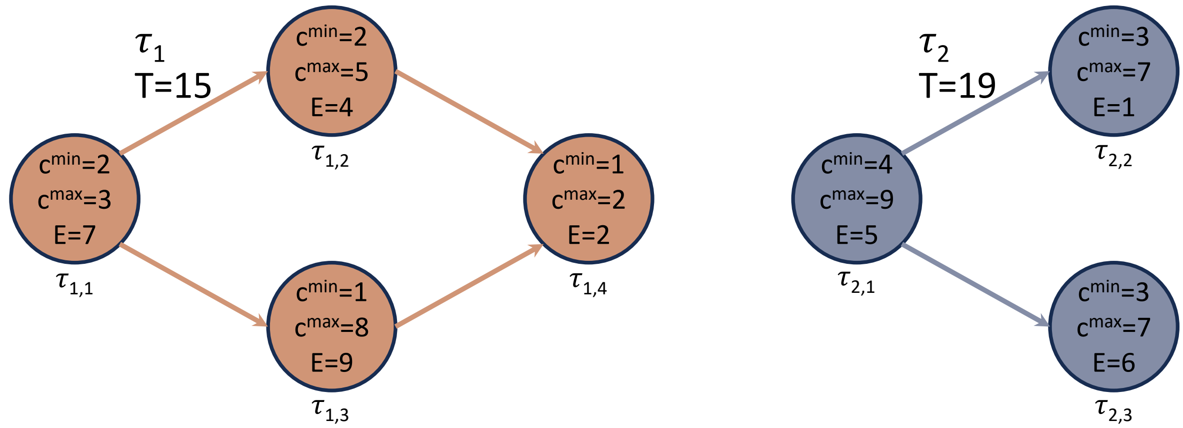


Figure 6.3: A running example using a system of two tasks with elastic subtasks.

Subtask Workloads

For each subtask $\tau_{i,j}$, define continuous variables $c_{i,j}$ representing the workload assigned to the subtask, and constrained as in Expression 6.5c:

$$c_{i,j}^{\min} \leq c_{i,j} \leq c_{i,j}^{\max} \tag{6.6}$$

Example 6. *For the tasks in Figure 6.3, we define* 7 *variables, constrained as follows:*

$$2 \leq c_{1,1} \leq 3 \quad 2 \leq c_{1,2} \leq 5 \quad 1 \leq c_{1,3} \leq 8 \quad 1 \leq c_{1,4} \leq 2$$

$$4 \leq c_{2,1} \leq 9 \quad 3 \leq c_{2,2} \leq 7 \quad 3 \leq c_{2,3} \leq 7$$

Objective

Our goal is to find an assignment of values to each variable $c_{i,j}$ that minimizes Expression 6.5a. The expression, when expanded, can be written as:

$$\sum_{\tau_{i,j}} \frac{1}{E_{i,j} T_i^2} \left(\left(c_{i,j}^{\max} \right)^2 - 2 c_{i,j}^{\max} c_{i,j} + c_{i,j}^2 \right)$$

Because our objective is to *minimize* this expression, and not to directly solve it, we can simplify by removing constant terms. Thus, our MIQP can be constructed so as to:

$$\textbf{minimize} \sum_{\tau_{i,j}} \left(\frac{1}{E_{i,j}T_i^2} \cdot c_{i,j}^2 - \frac{2c_{i,j}^{\max}}{E_{i,j}T_i^2} \cdot c_{i,j} \right) \tag{6.7}$$

Example 7. *For the tasks in Figure 6.3, we minimize the following objective:*

$$\frac{c_{1,1}^2}{7{\cdot}15^2} - \frac{2{\cdot}3{\cdot}c_{1,1}}{7{\cdot}15^2} + \frac{c_{1,2}^2}{4{\cdot}15^2} - \frac{2{\cdot}5{\cdot}c_{1,2}}{4{\cdot}15^2} + \frac{c_{1,3}^2}{9{\cdot}15^2} - \frac{2{\cdot}8{\cdot}c_{1,3}}{9{\cdot}15^2} + \frac{c_{1,4}^2}{2{\cdot}15^2} - \frac{2{\cdot}2{\cdot}c_{1,4}}{2{\cdot}15^2}$$

$$+\frac{c_{2,1}^2}{5{\cdot}19^2} - \frac{2{\cdot}9{\cdot}c_{2,1}}{5{\cdot}19^2} + \frac{c_{2,2}^2}{1{\cdot}19^2} - \frac{2{\cdot}7{\cdot}c_{2,2}}{1{\cdot}19^2} + \frac{c_{2,3}^2}{6{\cdot}19^2} - \frac{2{\cdot}7{\cdot}c_{2,3}}{6{\cdot}19^2}$$

Span

For each task τ_i, define a non-negative continuous variable L_i representing its span. It is required that L_i does not exceed T_i for τ_i to be schedulable, as $D_i = T_i$. Furthermore, a value of L_i exceeding T_i might result in the LHS of the expression in Equation 6.3 taking a negative value, which would be an inconsistent interpretation of the condition, and would result in an invalid solution. To enforce this, we add constraints of the form:

$$L_i \leq T_i \tag{6.8}$$

Example 8. *For the tasks in Figure 6.3, we add the following constraints for span:*

$$L_1 \leq 15 \quad L_2 \leq 19$$

Other variables and constraints to enforce the intended interpretation of each variable L_i as the span of τ_i are discussed further in Sections 6.4.2 and 6.4.3.

Processor Core Allocations

For each task τ_i, define a non-negative **integer** variable m_i representing the number of cores allocated to the task. This should be in sufficient number to guarantee schedulability according to Equation 6.2. To enforce this intended interpretation, we add constraints of the form:

$$m_i \geq \frac{\sum_j c_{i,j} - L_i}{T_i - L_i}$$

Since m_i is specified to be an integer variable, it will respect the ceiling operator that appears in Eqn. 6.2. Rearranging, this yields quadratic constraints of the form:

$$L_i + T_i{\cdot}m_i \geq m_i{\cdot}L_i + \sum_j c_{i,j} \tag{6.9}$$

This constraint, with the above constraint on span (Expression 6.8) will force L_i to remain strictly less than T_i, because

$$\lim_{L_i \to T_i^-} m_i = \infty$$

Example 9. *For the tasks in Figure 6.3, we add the following constraints for core allocations:*

$$L_1 + 15{\cdot}m_1 \geq m_1{\cdot}L_1 + c_{1,1} + c_{1,2} + c_{1,3} + c_{1,4}$$

$$L_2 + 19{\cdot}m_2 \geq m_2{\cdot}L_2 + c_{2,1} + c_{2,2} + c_{2,3}$$

Total Processor Cores

The total allocation of cores must not exceed m, the number available. To enforce this, we add the additional constraint:

$$\sum_i m_i \leq m \tag{6.10}$$

where m is a constant integer value.

Number of Constraints

Our MIQP has variables $c_{i,j}$ for each subtask $\tau_{i,j}$ and variables m_i and L_i for each task τ_i. Expression 6.10 represents a single constraint, though with numbers of terms linear in the total number of tasks. Expressions 6.8 and 6.9 both represent a constraint per task, with each constraint in Expression 6.9 having a number of terms linear in the number of subtasks of the corresponding task. Expression 6.6 represents a constraint for each subtask. The objective in Expression 6.7 has a number of terms linear in the total number of subtasks.

6.4.2 Task Span: A Constraint for Each Path

We now return to the problem of representing a DAG task's span in our MIQP. For a task τ_i, the variable L_i represents its span, which can be expressed as:

$$L_i = \max_{p_{i,k}} \left\{ \sum_{v_{i,j} \in p_{i,k}} c_{i,j} \right\}$$

over the set of paths $\{p_{i,k}\}$ between pairs of vertices in the task's representative DAG. To simplify this, we consider tasks τ_i for which the DAG has a single source vertex s and sink vertex t. Any task DAG τ_i, even those that are not weakly-connected, can be represented as a weakly-connected DAG with a single source s and sink t with the following construction. ① Add a vertex $v_{i,s}$ with execution time $c_{i,s} = 0$ and connect it with edges to all vertices in the DAG that do not already have incoming edges. Similarly, ② add a 0-workload vertex $v_{i,t}$, connected with edges from all vertices that do not already have outgoing edges. An example is illustrated in Figure 6.4.

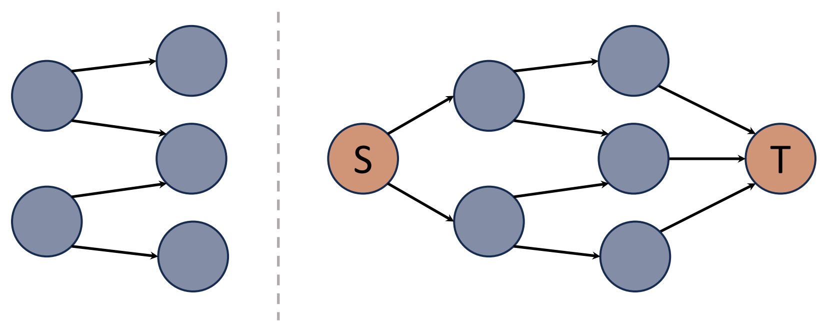


Figure 6.4: **Left:** a DAG with two source vertices and three sink vertices. **Right:** a unique source and unique sink vertex are added; if these both have workloads of 0, the corresponding task's execution remains unchanged.

Because of the restriction that $v_{i,a}$ is connected by an edge to $v_{i,b}$ only if $\tau_{i,b}$ directly succeeds $\tau_{i,a}$, every path from s to t might form the critical path, depending on the assignment of subtask execution times. Therefore, for each path $p_{i,k}$ from s to t, we add constraints of the form:

$$L_i \geq \sum_{v_{i,j} \in p_{i,k}} c_{i,j} \tag{6.11}$$

Example 10. *For the tasks in Figure 6.3, we add the following constraints for the span:*

$$L_1 \geq c_{1,1} + c_{1,2} + c_{1,4} \quad L_1 \geq c_{1,1} + c_{1,3} + c_{1,4}$$

$$L_2 \geq c_{2,1} + c_{2,2} \quad L_2 \geq c_{2,1} + c_{2,3}$$

Number of Constraints

The inequality in Expression 6.11 represents a constraint for every path from each task DAG's source vertex to its sink. The number of these constraints can therefore be expressed as the number of *maximal paths* through the DAG. It is shown in [115] that for a DAG without shortcuts[19] having $n = 3k$ vertices for $k \in \mathbb{N}$, the maximum number of maximal paths is 3^k. For our construction that adds a unique source and sink vertex, there can therefore be up to $3^{(n-2)/3}$ constraints, as illustrated in Figure 6.5.

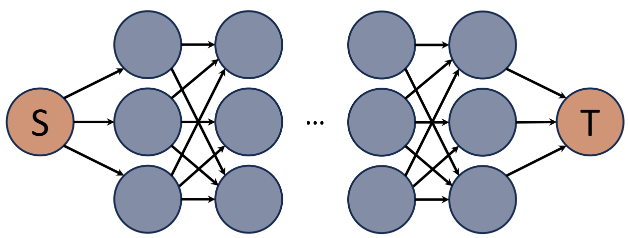


Figure 6.5: A DAG with $3^{(n-2)/3}$ maximal paths, each of which might be the critical path.

We refer the reader to [115] for a more detailed treatment of DAGs that do not have $3k$ vertices; suffice to say, the number of constraints may be exponential in the number of subtasks. We perform analysis for randomly-generated task DAGs in Section 6.6.

6.4.3 Task Span: A Polynomial Number of Constraints

To provide a smaller bound on the number of constraints in our MIQP, we propose an alternative method for enforcing the intended interpretation of the span variables L_i. From our prior work in [149], we introduce a term $l_{i,j}$ representing the span of the *subtask* $\tau_{i,j}$.

[19] An edge is a shortcut if the vertices it connects are connected by an alternate path; the requirement in Section 6.2.2 that two vertices are only connected by an edge if one directly succeeds the other eliminates shortcuts. A DAG without shortcuts is its own transitive reduction.

Definition 5 (Subtask Span). *The span $l_{i,j}$ of subtask $\tau_{i,j}$ in task τ_i represents the length of the longest path — weighted by the execution time of each subtask along the path — originating at the corresponding vertex $v_{i,j}$ of the task's DAG representation, including $v_{i,j}$ itself. This can be expressed by the following recurrence:*

$$l_{i,j} = c_{i,j} + \max_{v_{i,k}:\ v_{i,j} \prec v_{i,k}} \{l_{i,k}\} \tag{6.12}$$

For a task τ_i with a single source vertex s, this implies that $L_i = l_{i,s}$. From the above recurrence in Equation 6.12, we can enforce the intended interpretation of each span variable L_i by the following construction. For each subtask $\tau_{i,j}$ that does not correspond to a sink vertex in the task DAG, add a variable $l_{i,j}$ representing its span. The span of the source vertex[20] s is already represented by the variable L_i. Then for each such variable $l_{i,j}$, add the following constraint for every non-sink subtask $\tau_{i,k}$ that directly succeeds $\tau_{i,j}$:

$$l_{i,j} \geq c_{i,j} + l_{i,k} \tag{6.13}$$

For every subtask $\tau_{i,k}$ that directly succeeds $\tau_{i,j}$ and that *does* correspond to a sink vertex in the task DAG, add the following constraint instead:

$$l_{i,j} \geq c_{i,j} + c_{i,k} \tag{6.14}$$

Example 11. *For the tasks in Figure 6.3, we add variables L_1, $l_{1,2}$, $l_{1,3}$, and L_2 and constraints:*

$$L_1 \geq c_{1,1} + l_{1,2} \quad L_1 \geq c_{1,1} + l_{1,3}$$

$$l_{1,2} \geq c_{1,2} + c_{1,4} \quad l_{1,3} \geq c_{1,3} + c_{1,4}$$

$$L_2 \geq c_{2,1} + c_{2,2} \quad L_2 \geq c_{2,1} + c_{2,3}$$

[20] We assume — as we did in Section 6.4.2 — that a unique 0-workload source vertex s may be added if no unique source yet exists.

Number of Constraints

With this method, our MIQP has an additional variable for every subtask that does not correspond to a sink vertex or the unique source vertex of the corresponding task DAG. We add a constraint with 3 terms for every edge of the DAG; The maximum number of such constraints for a task with k subtasks is $\left\lfloor \frac{k^2}{4} \right\rfloor$, which follows from Turán's theorem [157].

Lemma 5. *Given a task τ with k subtasks and a directed acyclic graph G constructed from their precedence constraints as described in Section 6.2.2, the number of edges in G is at most*

$$\left\lfloor \frac{k^2}{4} \right\rfloor$$

Proof. Consider the corresponding undirected graph G^* constructed from G by replacing all directed edges with a corresponding undirected edge.

G^* contains no cliques of size 3. If it did, then without loss of generality, we can say it contains a clique consisting of vertices v_1, v_2, and v_3 connected by edges. In the corresponding directed graph G there are two possibilities:

1. Every vertex has one incoming and one outgoing edge. This forms a cycle, and G is acyclic, so this is a contradiction.
2. Some vertex has two incoming edges. Assume, without loss of generality, that it is v_3. For G^* to be a clique, then in G either v_1 has an outgoing edge to v_2 or vice versa; assume without loss of generality that $v_1 \to v_2$. But $v_2 \to v_3$ so the edge $v_1 \to v_3$ is a shortcut in G. This also is a contradiction.

From Turán's theorem [157], an undirected graph with k vertices that does not contain any cliques of size $r + 1$ has

$$\left(1 - \frac{1}{r}\right) \frac{k^2}{2}$$

as an upper bound on the number of edges. Because G^* has no cliques of size 3, $r = 2$, so this upper bound is $k^2/4$. Since this must take an integer value, it can be expressed as $\left\lfloor \frac{k^2}{4} \right\rfloor$. □

6.5 Joint Compression with Dynamic Programming

We now present an alternative approach to the problem of workload compression using an MIQP. Rather than constructing a joint problem over all tasks, the idea is to construct an MIQP for each task *individually*, then solve to find the optimal assignment of subtask workloads (and the corresponding objective value) for each possible core allocation. This defines a set of discrete states for each task corresponding to different core allocations; the optimal assignment *overall* can then be determined using a similar dynamic programming (DP) technique to the approach of Orr et al. in [121].

6.5.1 Motivation

Compressing task workloads individually, then solving the joint problem with DP, has three advantages over the joint MIQP presented in Section 6.4.

Faster Solution Search

We observed in Section 4.8 that solving an MIQP separately for each individual task, rather than a single joint MIQP for the complete task system, is significantly faster for elastic scheduling of constrained-deadline, fixed-priority sequential tasks.

This same idea also applies to subtask-level elastic scheduling. Though we cannot make theoretical guarantees about improved complexity, we empirically demonstrate in Section 6.6.3 that in many cases, solving multiple MIQPs for each individual task, then constructing a dynamic program to optimally allocate cores to all tasks, is faster than just solving a single joint MIQP.

Efficient Admission Control

Buttazzo's elastic scheduling model in [39, 40] is not simply intended for adjusting a predefined set of tasks to be schedulable on a resource-constrained system. Its primary use-case is in dynamic and open systems where the set of active tasks may change, and therefore an efficient approach to admission control is desirable. Indeed, Chapter 2 of this dissertation demonstrates an algorithm that achieves elastic task compression for a uniprocessor in time quasilinear in the number of tasks, but enables admission control in only linear time.

Given that task parameters (control-flow DAGs, execution times, deadlines, etc.) are assumed to be characterized offline, it is also reasonable to think that discrete states corresponding to optimal subtask-level workload compression for different core allocations could also be computed offline. Then, when configuring the system for a set of tasks, or during admission control of a new task, only the pseudo-polynomial DP problem needs to be solved.

Joint Compression of Low-Utilization Tasks

Finally, as we will show, a DP-based approach allows us to also address federated scheduling of low-utilization tasks that must execute concurrently with high-utilization parallel tasks. We can compute the amount of compression needed to schedule the complete set of low-utilization tasks for each possible core allocation, then determine the corresponding objective value for each state using Expression 6.5a. These states are then considered jointly by the dynamic program with those for each high-utilization parallel task; the solution determines the number of cores to allocate to each high-utilization task, and how many to allocate jointly to the low-utilization tasks.

6.5.2 Method

The method described above is realised in two steps:

1. Construct and solve an MIQP for each task individually over every possible core allocation.

2. Construct an instance of a multiple-choice knapsack problem to allocate cores to each task such that **(a)** the objective in Expression 6.5a is minimized while **(b)** the total allocation of cores does not exceed the number available. This approach is outlined in Algorithm 12, which takes a set Γ of n tasks to be scheduled on m processor cores.

Algorithm 12: COMPRESS-QP(Γ,m)

1 **Input:** A set Γ of n high-utilization parallel tasks, m available processor cores
2 **Output:** A set $\{c_{i,j}\}$ of subtask workload assignments
3 ▷ `Find optimal state for each core allocation`
4 **forall** $\tau_i \in \Gamma$ **do**
5 $\quad C_i^{\min} \leftarrow \sum_j c_{i,j}^{\min}$, $C_i^{\max} \leftarrow \sum_j c_{i,j}^{\max}$
6 $\quad L_i^{\min} \leftarrow$ Compute span according to $c_{i,j}^{\min}$ values
7 $\quad L_i^{\max} \leftarrow$ Compute span according to $c_{i,j}^{\max}$ values
8 $\quad m_i^{\min} \leftarrow \left\lceil \frac{C_i^{\min} - L_i^{\min}}{T_i - L_i^{\min}} \right\rceil$, $m_i^{\max} \leftarrow \left\lceil \frac{C_i^{\max} - L_i^{\max}}{T_i - L_i^{\max}} \right\rceil$
9 $\quad$ **forall** $m_{i,k} \leftarrow m_i^{\min}..(m_i^{\max}-1)$ **do**
10 $\quad\quad$ Construct and solve an MIQP to obtain optimal subtask workloads and corresponding objective value $O_{i,k}$ to compress the *single* task τ_i to execute on $m_{i,k}$ cores.
11 ▷ `Find optimal joint state for` m `cores`
12 **if** $\sum_i m_i^{\max} \leq m$ **then return** *No compression needed*
13 **if** $\sum_i m_i^{\min} > m$ **then return** *Not schedulable*
14 ▷ `Adapted multiple-choice knapsack`
15 $DP[0..m][0..n]$▷ `Table to track optimal solution.`
16 $DP[0][*].O \leftarrow \infty$, $DP[*][0].O \leftarrow \infty$
17 **for** $m^* \leftarrow 1..m$ **do**
18 $\quad$ **for** $i \leftarrow 1..n$ **do**
19 $\quad\quad MIN \leftarrow \infty$, $ALLOC \leftarrow -1$
20 $\quad\quad$ **for** $m_{i,k} \leftarrow m_i^{\min}..\min(m_i^{\max}, m^*)$ **do**
21 $\quad\quad\quad$ **if** $i = 1$ **then**
22 $\quad\quad\quad\quad MIN \leftarrow O_{i,k}$
23 $\quad\quad\quad\quad ALLOC \leftarrow m_{i,k}$
24 $\quad\quad\quad$ **else if** $DP[m^* - m_{i,k}][i-1].O + O_{i,k} < MIN$ **then**
25 $\quad\quad\quad\quad MIN \leftarrow DP[m^* - m_{i,k}][i-1].O + O_{i,k}$
26 $\quad\quad\quad\quad ALLOC \leftarrow m_{i,k}$
27 $\quad\quad$ **if** $ALLOC > -1$ **then**
28 $\quad\quad\quad DP[m^*][i].M \leftarrow DP[m^* - ALLOC][i-1].M$
29 $\quad\quad\quad DP[m^*][i].M.insert(ALLOC)$
30 $\quad\quad\quad DP[m^*][i].O = MIN$
31 $\quad\quad$ **else** $DP[m^*][i] = DP[m^* - 1][i]$
32 **return** $DP[m][n]$

1. Constructing and Solving MIQPs

For each individual task τ_i, we compute the minimum $m_i^{\min}$ and maximum $m_i^{\max}$ number of cores that it can be allocated. For any allocation less than the minimum, τ_i is not guaranteed to be schedulable; any allocation greater than the maximum is wasted capacity. These are computed in lines 5–8 of Algorithm 12.

Then for each possible core allocation m^* in the range $[m_i^{\min}, m_i^{\max}-1]$, we construct and solve an MIQP according to the procedure in Section 6.4 for just the *individual* task.

The MIQP may be simplified by removing the variable m_i that represents the number of cores assigned to task τ_i and replacing it instead with a constant $m = m^*$. In doing so, the constraint taking the form of Expression 6.9 becomes linear instead of quadratic, and the constraint of Expression 6.10 is removed.

Solving for each value of m^* in this way gives us a set of optimal subtask workload assignments and objective function values for each allocation; for $m^* = m_i^{\max}$, every subtask is assigned as its workload $c_{i,j} = c_{i,j}^{\max}$ and the task's contribution to the objective function in Expression 6.5a is 0.

2. Joint Allocation as a Multiple-Choice Knapsack Problem

Lines 4–12 of Algorithm 12 give us, for each task τ_i, a group of pairs of *weight* (processor core allocation, $m_{i,k}$) and *cost* (the minimum value taken by Expression 6.5a, $O_{i,k}$) values for each $m_{i,k} \in [m_i^{\min}, m_i^{\max}]$. (For $m_i^{\max}$, the cost is 0.) The goal is to select a pair from each group that minimizes the total cost, while preventing the total weight from exceeding the number of available cores m. As shown in [121], this can be reduced to an instance of the multiple-choice knapsack problem, for which a pseudo-polynomial DP-based algorithm is presented in [81]. Our problem differs slightly from multiple-choice knapsack because the goal is to *minimize* total cost, rather than *maximize* total profit. Lines 15–32 of Algorithm 12 in this paper are adapted from [121, Alg. 1], which solves the problem for elastic scheduling with discrete execution states. In our case, rather than execution states, we consider discrete core allocations. And unlike [121, Alg. 1], we demonstrate how to track the selected core allocations to assign subtask workloads after the algorithm completes.

Our algorithm builds a two-dimensional table DP where $DP[m^*][i]$ gives the optimal solution after considering the first $i \leq n$ tasks on $m^* \leq m$ cores. Each entry in the table is a pair $\langle M, O \rangle$ where M is a set that tracks the number of cores allocated to those i tasks, and O is the corresponding minimum objective function value. It first assigns a score of infinity to the impossible case of scheduling on 0 cores and the trivial case of scheduling 0 tasks. The algorithm then considers (line 17) scheduling tasks on m^* CPUs, considering the first i tasks (line 18).

For each value of $m_{i,k} \in [m_i^{\min}, m_i^{\max}]$, the algorithm checks whether allocating $m_{i,k}$ cores to task τ_i improves the result (i.e., decreases MIN). For τ_1, as there are no cores allocated yet to other tasks, the algorithm simply sets MIN to the best objective $O_{i,k}$ for $m_{i,k}$ cores (lines 21–23). For each remaining task τ_i, the algorithm considers the joint allocation of $m_{i,k}$ cores to τ_i and $m^*{-}m_{i,k}$ cores to the previous tasks (lines 24–26). If an improved allocation is found, then the entry of the DP table corresponding to m^* cores and the first i tasks is updated (lines 27–30) to track the current best core allocation for those tasks and the corresponding objective. Otherwise, it is updated to match the best allocation over the previously-considered $m^*{-}1$ cores.

The algorithm returns the pair stored in the entry of DP corresponding to scheduling the complete set of n tasks on all m. For each task, subtask workloads can then be assigned according to the corresponding MIQP solution for m_i cores.

Runtime Complexity and Admission Control

While we cannot make guarantees about the time to solve each MIQP, the DP portion of Algorithm 12 is pseudopolynomial in n and m. There are m CPUs to allocate (line 17) to n tasks (line 18). For each task τ_i, we consider allocations from $m_i^{\min}$ to $m_i^{\max}$, stopping if the currently-considered allocation m^* is reached (line 19); this bounds the number of iterations of the inner **for** loop to m, since $m^*{\le}m$. The total worst-case running time is therefore $\boldsymbol{\Theta(n{\cdot}m^2)}$. As justified in Section 6.5.1, if the optimal set of task workloads for each core allocation are obtained *offline* when a task's other parameters are characterized, then admission of a new task to an already-compressed system can be achieved by executing lines 14–32 of the algorithm, enabling bounded-time admission control. We evaluate this in the context of synthetically-generated parallel tasks in Section 6.6.

6.5.3 Joint Scheduling of Low-Utilization Tasks

Our DP-based approach to subtask-level elasticity also enables joint scheduling of low-utilization tasks. The key idea is that we can consider $m_{\text{low}}^{\min}$ and $m_{\text{low}}^{\max}$ as the number of cores necessary to schedule the complete set Γ_{low} of low-utilization tasks when fully compressed versus uncompressed. For every $m^* \in [m_{\text{low}}^{\min}, m_{\text{low}}^{\max} - 1]$, we can quantify the amount of compression necessary to achieve schedulability on m^* cores. By then solving for the corresponding objective function value in Expression 6.5a for the compressed Γ_{low}, we obtain a set of discrete core assignments and costs. This allows the complete set Γ_{low} to be integrated into the DP-based algorithm as if it were a single high-utilization parallel task.

Obtaining values $m_{\text{low}}^{\min}$ and $m_{\text{low}}^{\max}$, as well as the amount of compression necessary to achieve schedulability on m^* cores, depends on the multiprocessor scheduling algorithm used. While complete coverage of multiprocessor scheduling is outside the scope of this dissertation, we outline how the approaches in Chapter 3 for fluid and partitioned EDF scheduling can be applied in this context.

Fluid Scheduling

Recall from Section 3.2.1 that under the fluid scheduling paradigm, individual tasks are assigned a fraction f of a processor at each instant in time. This is a convenient abstraction that considers a set Γ of tasks τ_i to be schedulable on m cores so long as **(a)** the total utilization $\sum_i U_i$ of Γ does not exceed m, and **(b)** the individual utilizations U_i of each task τ_i do not exceed 1 [19].

For low-utilization tasks, condition **(b)** is automatically satisfied. We can therefore obtain $m_{\text{low}}^{\min}$ and $m_{\text{low}}^{\max}$ as:

$$m_{\text{low}}^{\min} = \left\lceil \sum_i U_i^{\min} \right\rceil \quad m_{\text{low}}^{\max} = \left\lceil \sum_i U_i^{max} \right\rceil \tag{6.15}$$

where $U_i^{\min} = C_i^{\min}/T_i$ or $C_i/T_i^{\max}$ (similarly for $U_i^{\max}$), depending on whether τ_i is computationally-elastic or rate-elastic.

Then for $m^* \in [m_{\text{low}}^{\min}, m_{\text{low}}^{\max} - 1]$, we assign values U_i to each task τ_i that satisfy the conditions under Buttazzo's elastic model [39, 40] — described in Section 2.2.2 — with the desired utilization U_D equal to m^*.

The total execution time (on top of the $\Theta((n_{\text{high}} + 1)\cdot m^2)$ to solve the DP problem jointly with n_{high} high-utilization parallel tasks) can be kept to a minimum by using Algorithm 2 in Section 2.3. Computing $m_{\text{low}}^{\min}$ and $m_{\text{low}}^{\max}$ can be done in time linear in n_{low}, the number of tasks in Γ_{low}. Compressing to m^* cores can be done in time $\mathcal{O}(n_{\text{low}} \cdot \log(n_{\text{low}}))$. However, we remind the reader that the quasilinear overhead is due to the initial step of sorting the list of tasks according to their ϕ_i values. This ordering does not depend on the desired utilization U_D. Therefore, this can be done only once; it does not have to be repeated for each additional value m^*. The remainder of Algorithm 2 takes time $\mathcal{O}(n_{\text{low}})$.

Thus, the worst-case running time is $\boldsymbol{\Theta(n_{\text{low}} \cdot \log(n_{\text{low}}) + m{\cdot}n_{\text{low}})}$ for the initial sort, followed by at most m linear-time invocations of Algorithm 2, since we can stop when m^* exceeds m, the total number of cores available. Computing Expression 6.5a for each m^* is also linear in n_{low}.

Partitioned Scheduling

While fluid scheduling is a convenient abstraction, and implementations exist to approximate it [41], it often remains impractical in real systems [118]. A more practical scheduling paradigm is partitioned scheduling, where tasks are distributed to processors a priori, then scheduled with other tasks on that processor according to a common approach (e.g., fixed-priority or EDF). An optimal distribution for partitioned EDF is equivalent to the bin-packing problem, and is therefore NP-hard in the strong sense, but approximation algorithms exist that provide guaranteed schedulability if a utilization bound is not exceeded [12]. For example, as discussed in Section 3.4, a set of low-utilization tasks are schedulable on m^* processor cores under partitioned EDF using a first-fit or best-fit packing if their total utilization does not exceed $(m^* + 1)/2$.

For joint compression of low-utilization tasks, the approach outlined above for fluid scheduling can be adopted as follows. First, we obtain $m_{\text{low}}^{\min}$ and $m_{\text{low}}^{\max}$. We observe that $U_{\text{SUM}}^{\min} \leq (m_{\text{low}}^{\min} + 1)/2$ — and similarly for $U_{\text{SUM}}^{\max}$ and $m_{\text{low}}^{\max}$ — so:

$$m_{\text{low}}^{\min} = \left\lceil 2 \sum_i (U_i^{\min}) - 1 \right\rceil \quad m_{\text{low}}^{\max} = \left\lceil 2 \sum_i (U_i^{\max}) - 1 \right\rceil \tag{6.16}$$

For $m^* \in [m_{\text{low}}^{\min}, m_{\text{low}}^{\max} - 1]$, we can again use Algorithm 2 to compress to a desired utilization $U_D = (m^* + 1)/2$.

The worst-case running time is still $\boldsymbol{\Theta(n_{\mathrm{low}} \cdot \log(n_{\mathrm{low}}) + m{\cdot}n_{\mathrm{low}})}$, as the procedure is equivalent to that of fluid scheduling, but with different utilization bounds. Once a core allocation has been obtained by solving the DP problem, partitioning the low-utilization tasks using a first-fit bin packing requires at most $\Theta(n_{\mathrm{low}} \cdot \log(n_{\mathrm{low}}))$ time.

An alternative method — one that schedules more optimistically and applies less compression, but which has a higher execution time complexity — arises from our BS-Order implementation of elastic scheduling for partitioned EDF in Chapter 3. For a given number of cores m^*, we can perform binary search for the minimum amount of compression λ (see Equation 6.4) needed to schedule the set of low-utilization tasks on those cores according to the best-fit or first-fit bin packing heuristics.

We would again quantify the amount of compression, and corresponding objective value, for each number of cores $m^* \in [m_{\mathrm{low}}^{\mathrm{min}}, m_{\mathrm{low}}^{\mathrm{max}} - 1]$. However, by using heuristic bin packing, we do not know $m_{\mathrm{low}}^{\mathrm{min}}$ and $m_{\mathrm{low}}^{\mathrm{max}}$ a priori. We do know that $m_{\mathrm{low}}^{\mathrm{min}}$ is bounded below by $\lceil \sum_i U_i^{\mathrm{min}} \rceil$ and $m_{\mathrm{low}}^{\mathrm{max}}$ is bounded above per Equation 6.16.

For m total processors, partitioned EDF elastic scheduling has to be invoked up to m times. Given the desired granularity ϵ of the search for the amount of compression λ, the worst-case running time of this approach is therefore

$$\boldsymbol{\Theta\left(m \cdot (n \log n + n \cdot m) \cdot \log\left(\frac{\lambda^{\max}}{\epsilon}\right)\right)}$$

per Expression 3.3.

6.6 Evaluation

In this section, we empirically evaluate the performance of the algorithms detailed in Sections 6.4 and 6.5 for synthetically-generated task sets, and compare them to the prior state of the art in [119, Algorithm 1].

6.6.1 Analysis of Span Constraints

We begin by analyzing the number of constraints necessary to enforce the intended interpretation of the span variables in the MIQP discussed in Sections 6.4.2 and 6.4.3. Though we have already provided theoretical upper bounds, we would like to now empirically quantify a range of realistic problem sizes associated with sets of synthetically-generated DAG tasks.

Experimental Setup

We generate DAGs according to a modified version of the Erdős-Rényi method [43]:

1. Select a number of vertices k for the DAG G (we iterate over values of k from 5–50).
2. For each pair of vertices in $\{v_2, ..., v_{k-1}\}$, a connecting edge is added with probability p (we iterate over values of p from 0.05–0.95 in steps of 0.05). The edge is always directed from the smaller to larger vertex index to guarantee the graph remains acyclic.
3. Vertex v_1 is the source vertex: direct an edge from it to all remaining vertices (except v_k) with no incoming vertices. Similarly, vertex v_k is the sink: direct an edge to it from all vertices with no outgoing vertices. This guarantees that the DAG is weakly connected.
4. For every edge E connecting vertex v_a to v_b, if there exists a path from v_a to v_b in $G \backslash E$, then E is a shortcut and is removed as illustrated in Figure 6.6. This guarantees that no path from source to sink is a subset of another path, so every path might form the critical path, depending on its vertex weights (i.e., the corresponding subtask execution times).

For each value of k and p, we randomly generate 10 000 graphs.

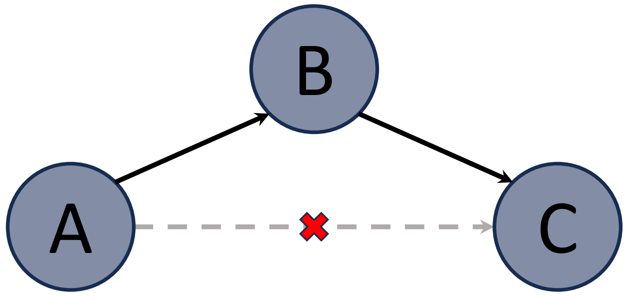
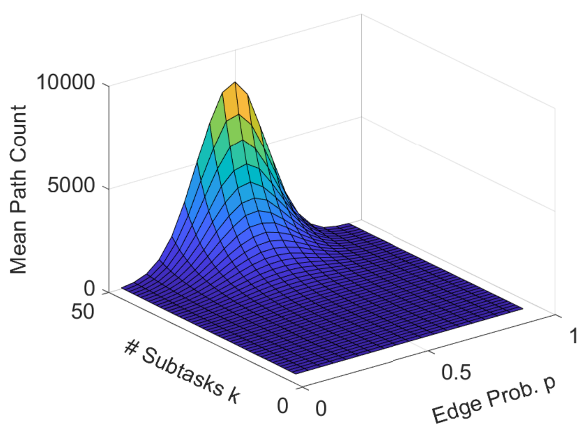


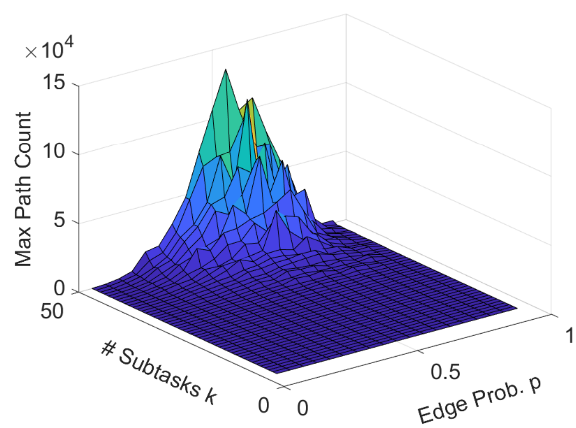
Figure 6.6: Removing shortcut edges.

Counting Maximal Paths

For each DAG, we count the number of paths from the source to the sink vertex; these are exactly the set of maximal paths and correspond to constraints in the form of Expression 6.11 in Section 6.4.2. We then calculate the mean and maximum count for each pair (k, p). Results are plotted in Figure 6.7.



(a) Mean path counts.



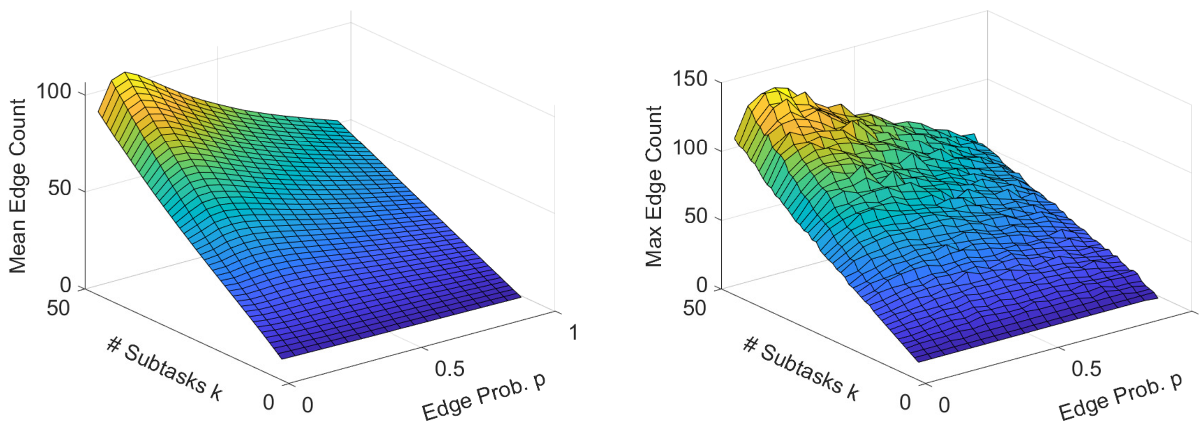
(b) Max path counts.

Figure 6.7: Maximal Path Counts

We observe that an edge probability of 0.5 is expected to produce the largest number of maximal paths. For tasks with 50 subtasks and $p = 0.5$, 8465 constraints of the form of Expression 6.11 will be added on average with a maximum observed of 106 560. However, an edge probability of 0.55 gives the maximum observed overall at 133 632 such paths. In comparison, the maximum possible for the pathological case illustrated in Figure 6.5 is $3^{(50-2)/3} = 3^{16}$, which is over 43 million.

Counting Edges

For each DAG, we also count the number of edges remaining after removal of shortcut edges. Each such edge corresponds to a constraint in the form of Expression 6.13 or Expression 6.14 in Section 6.4.3. Results are plotted in Figure 6.8.



(a) Mean edge counts. (b) Max edge counts.

Figure 6.8: Edge Counts

We observe that for smaller numbers of subtasks, an edge probability of 0.2 is expected to produce the largest number of edges, as edge shortcuts are removed after the initial set of edges are generated. As the number of subtasks k approaches 50, $p = 0.15$ is expected to result in the most edges: 106 on average. The maximum observed overall was 136. ***For tasks with fewer subtasks, adding a constraint per path typically yields fewer constraints, but as the number of subtasks increases, the number of paths rapidly overtakes the number of edges.***

6.6.2 MIQP Solver Performance

We now evaluate the feasibility of using an off-the-shelf MIQP solver to assign execution times to subtasks according to the optimization problem listed in Expression 6.5.

Implementation

We formulate an MIQP for each task according to the procedure in Section 6.4. We use a custom C++ wrapper into which we link version 10.0.3 [105] of the Gurobi Optimizer [72] to execute the MIQP. Each task is represented as a data structure (`struct`) with subtask workloads $c_{i,j}$ and constraints $c_{i,j}^{\min}$, $c_{i,j}^{\max}$ on those workloads, stored as arrays (`std::vector`) of single-precision floating-point representations. Task deadlines D_i, which are assumed equal to the period T_i, are stored as integers. DAG edges representing the precedence constraints among subtasks are encoded in an adjacency matrix.

We quantify execution time performance by reading from the standard library's high resolution clock (`std::chrono::high_resolution_clock`). Compilation is performed using the Gnu Compiler Collection (GCC) at optimization level `O3`. We enclose calls to the solver between calls to `std::atomic_signal_fence` using sequentially-consistent ordering; this avoids instruction reordering around clock reads. We evaluate execution times in a single thread on a server with an AMD EPYC 9754 and 128GB of RAM running Linux 5.14.0. Simultaneous Multithreading and CPU throttling are disabled.

Compressing Individual Tasks

We begin by randomly generating tasks according to the modified Erdős-Rényi method outlined above, using an edge probability of $p = 0.5$ since we have observed that this typically induces the greatest number of maximal paths. For each value k (number of subtasks) in 5–50, we generate 1000 such tasks, for a total of 46 000.

Each subtask $\tau_{i,j}$ has its elasticity $E_{i,j}$ randomly selected as an integer from the range 1–100. To assign a range of acceptable execution times to each subtask, we randomly select two integer values in the range 1–100. The smaller value is assigned to $c_{i,j}^{\min}$ and the larger to $c_{i,j}^{\max}$. So that the task remains high-utilization even if all subtasks are assigned their minimum execution times, we randomly select D_i as an integer from the range $[L_i^{\max} + 1, C_i^{\min} - 1]$ (if $D \leq L_i$, the core assignment in Equation 6.2 becomes invalid). If for some task τ_i, $L_i^{\max} + 1 > C_i^{\min} - 1$, values of $c_{i,j}$ are regenerated.

Generated parameters are used with Equation 6.2 to determine the minimum $m_i^{\min}$ and maximum $m_i^{\max}$ number of cores to guarantee schedulability for each task; the integer value m of total cores available is selected uniformly from the range $[m_i^{\min}, m_i^{\max} - 1]$.

We formulate two MIQPs for each task according to the procedure in Section 6.4, using both proposed methods to enforce the intended interpretation of the span variables (with a constraint for each maximal path per Section 6.4.2 or with a constraint for each edge per Section 6.4.3). We then solve using Gurobi according to the above implementation details. The solver is configured to execute in a single thread, which allows us to run separate instances of the algorithm sequentially on 100 of the unused physical cores on our server, splitting up the work of compressing all 46 000 considered task systems. Results are plotted in Figure 6.9.

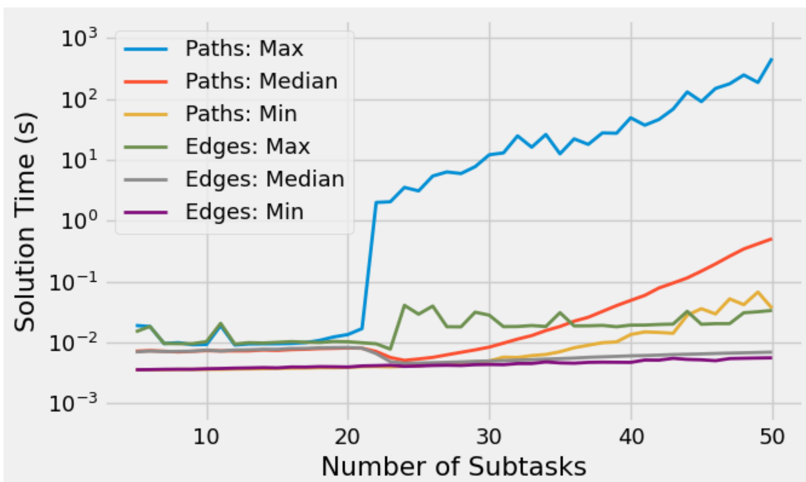


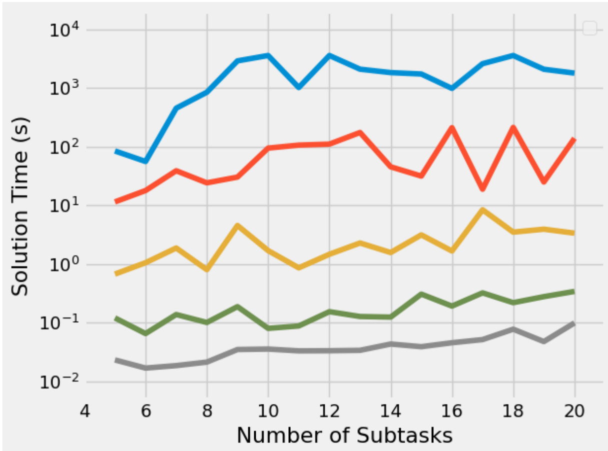
Figure 6.9: MIQP times for individual tasks.

We observe that for smaller numbers of subtasks (i.e., up to around 20), the selection of the method for enforcing the intended interpretation of the span variables does not have a significant impact on execution time. However, ***as the number of subtasks increases further, a constraint per edge becomes significantly faster***. With a constraint per edge, a solution was reached in under 41 ms in the worst case, with a median under 6.0 ms for 50 subtasks. But with a constraint per path, solution search took up to 462 s in the worst case (1127× slower), and for tasks with 50 subtasks, the median time was 505 ms (72.6× slower).

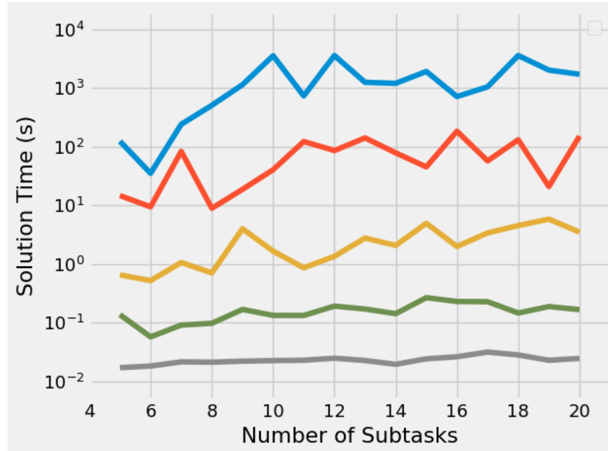
Joint Task Compression

We next consider the joint compression of multiple parallel tasks. We randomly generate task sets of size n from 2–10 in steps of 2. Every task in a task set is assigned the same number k of subtasks; for each value n, we consider values of k from 5–20. For each pair (n, k), we generate 100 task sets using the above methodology, for a total of 8000. Each task DAG again has an edge probability of $p = 0.5$. Generated parameters are again used with Equation 6.2 to determine the minimum $m^{\min}$ and maximum $m^{\max}$ number of cores to guarantee schedulability for each task system; the integer value m of total cores available is selected uniformly from the range $[m^{\min}, m^{\max} - 1]$.

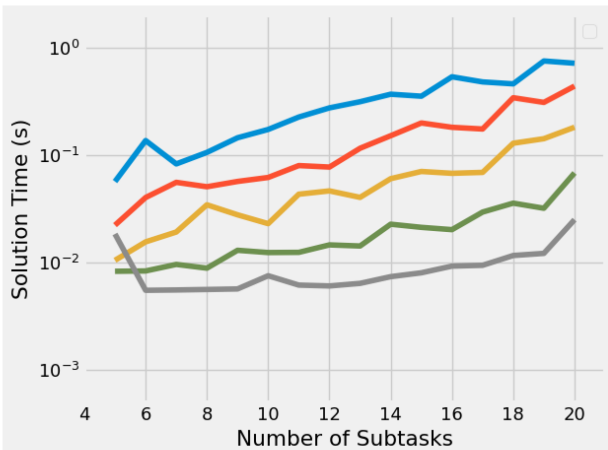
We again formulate two MIQPs for each task according to the procedures in Section 6.4, then solve with Gurobi using a single thread, splitting the work among 100 physical cores on our server. This time, we force the solver to terminate if no solution is found after one hour. Results are plotted in Figure 6.10.



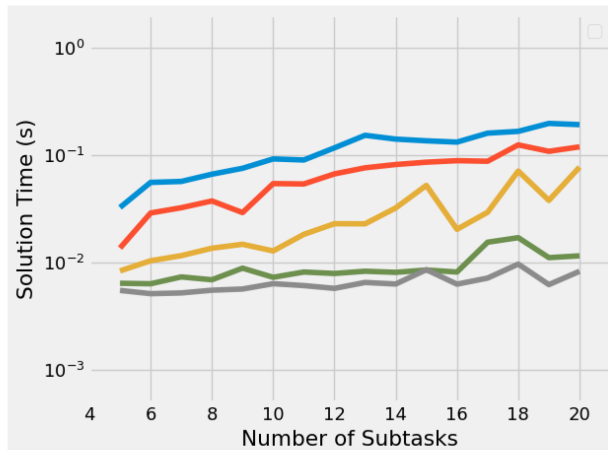
(a) Span Constraint per Path: Max Times



(b) Span Constraint per Edge: Max Times



(c) Span Constraint per Path: Median Times



(d) Span Constraint per Edge: Median Times

Figure 6.10: MIQP times when solved jointly for multiple tasks. Series in each plot, from bottom to top, are for sets of 2, 4, 6, 8, and 10 tasks.

From these plots, we make the following observations:

- From comparing the plots on the left and right sides of the figure, no substantial difference in maximum execution times is observed between the two methods for enforcing the intended interpretation of the span variables.[21] This is unsurprising, as Figure 6.9 did not illustrate a significant difference for single tasks of up to 20 subtasks. However, it is noteworthy that this similarity is preserved when compressing jointly for up to 10

[21] There is some difference — with a span constraint per path, the maximum observed execution time for 2 tasks was 98.9 ms, 3.15× slower than the maximum 31.5 ms observed with a span constraint per edge — but this is not visually obvious due to the logarithmic scale in the y-axis.

tasks. Moreover, we *do* see that for more than 2 tasks, the median execution times associated with maximal path constraints are higher than those associated with edge constraints.

- Maximum execution times occasionally reach one hour when jointly compressing 10 tasks. In fact, of the 16 000 tested MIQPs, 6 timed out at the one hour limit we imposed. Of those, 4 enforce span with a constraint for each maximal path, and have 10, 12, 12, and 18 subtasks per task. The other 2 enforce span with a constraint for each edge, and have 12 and 18 subtasks per task.

- Most importantly, it comes as no surprise that ***execution times increase rapidly as tasks are added.*** For every two additional tasks beyond the first 4, both the median and maximum execution times increase by about an order of magnitude. With a span constraint per path, the maximum observed execution time for 2 tasks was only 98.9 ms, at least 36000× faster than the one hour timeout for 10 tasks. And with a span constraint per edge, the maximum observed execution time for 2 tasks was only 31.5 ms, over 114 000× faster than for 10 tasks.

This suggests that our dynamic-programming approach proposed in Section 6.5 may be more efficient with larger numbers of tasks. Recall that its worst-case running time for n tasks on m cores is $\Theta(n{\cdot}m^2)$. Given our methodology for generating sets of parallel tasks, the number of cores in expectation should scale proportionally to the number of tasks, and so the running time of the DP should scale approximately with n^3. From 2 to 10 tasks, we therefore expect the execution time associated with the DP to increase by about 125×, not the $\sim 10^5\times$ observed for the joint MIQP. We test this hypothesis in the next subsection.

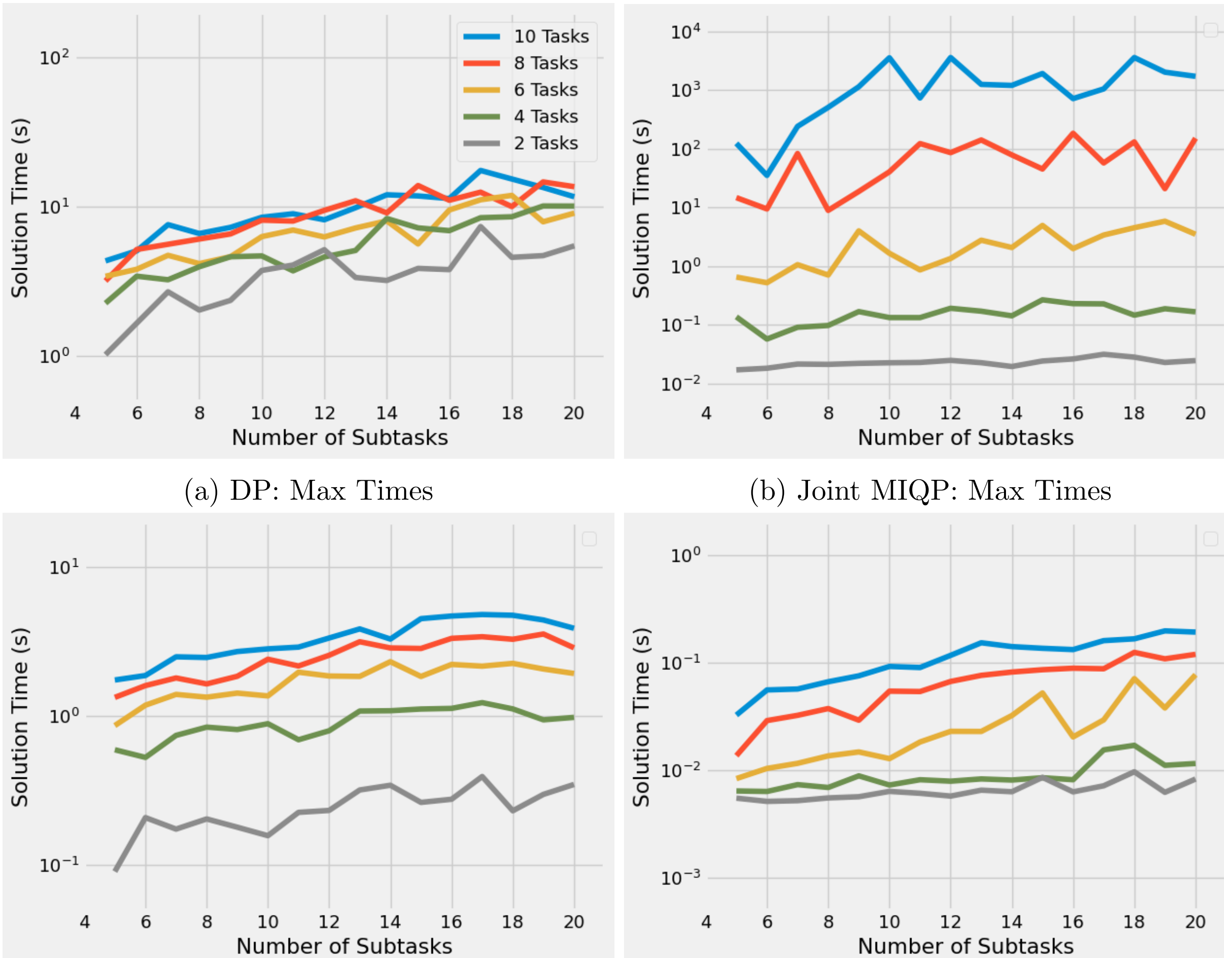
6.6.3 DP-Based Solution Performance

We have shown that solving the MIQP jointly for larger sets of tasks rapidly becomes infeasible in a short amount of time. However, we've also shown that solving the MIQP for an *individual* task is relatively efficient, typically on the order of a few milliseconds even for large numbers of subtasks (up to 50). We therefore expect our DP-based approach of Section 6.5 to outperform the single MIQP for larger sets of tasks.

Side-By-Side Comparison

We begin by evaluating our DP-based approach using the same sets of tasks that we generated to test the performance when solving a single joint MIQP over every task, with 100 sets of tasks for each number n of tasks from 2–10 in steps of 2 and each number k of subtasks from 5–20. The MIQP for individual tasks represents span variables according to the methodology in Section 6.4.3, using a constraint for each DAG edge, since this was already shown to be more efficient than using a constraint for each maximal path.

As before, we split the work among 100 threads on our system. We separately measure the execution time associated with solving MIQPs to obtain discrete compression states for each task associated with each core assignment, and the execution time to subsequently solve the dynamic program. Figure 6.11 compares the total execution time with that of solving a single MIQP jointly for all tasks with a span constraint per edge; Figures 6.11b and 6.11d are repeated from Figure 6.10 to provide a side-by-side comparison.



(a) DP: Max Times

(b) Joint MIQP: Max Times

(c) DP: Median Times

(d) Joint MIQP: Median Times

Figure 6.11: Total execution times to solve a joint MIQP with a span constraint per edge versus using the DP-based approach.

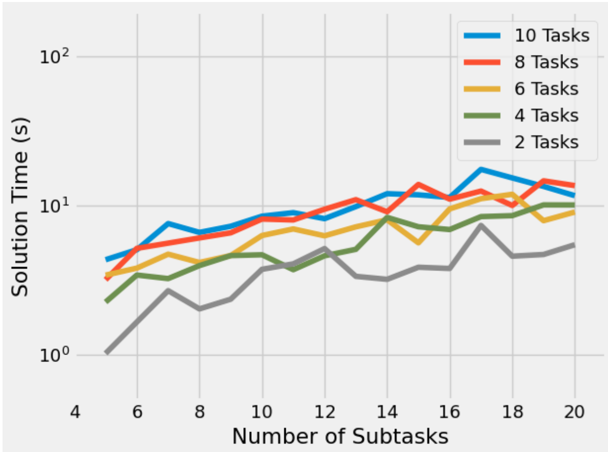
This allows us to draw the following conclusions about the DP-based approach.

- ***For a small number of tasks, solving a single joint MIQP is more efficient.*** For sets of 2–4 tasks, the maximum observed execution time for each number of subtasks remains under 1 second when solving a joint MIQP, but is consistently over 1 second (reaching as high as 10 seconds) for the DP-based approach. This makes sense, because the DP-based approach has to solve several individual MIQPs for each task — one for each possible core assignment. As we will show, this dominates its execution time.

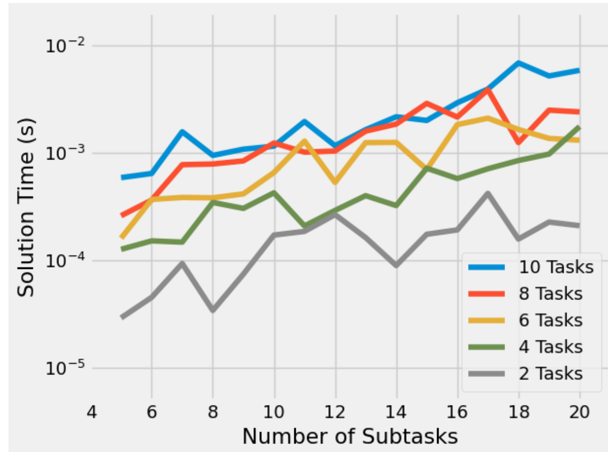
- However, ***as the number of tasks increases, the execution time associated with the DP-based approach grows more slowly than that of the single joint MIQP.*** For the DP, the maximum observed execution time for 2 tasks is 7.39 s, compared to 17.6 s for 10 tasks; this is only a 2.39× increase. Compared to the $\sim 10^5\times$ increase in execution time from 2 to 10 tasks observed for the joint MIQP, the DP-based approach scales much more slowly.

- As a result, ***for larger numbers of tasks, the maximum execution time associated with the DP-based approach is much faster than that of the single MIQP.*** While we cannot report an exact speedup, since the joint MIQP timed out after 1 hour, the execution time of the DP-based approach is at least 205× faster in the worst case.

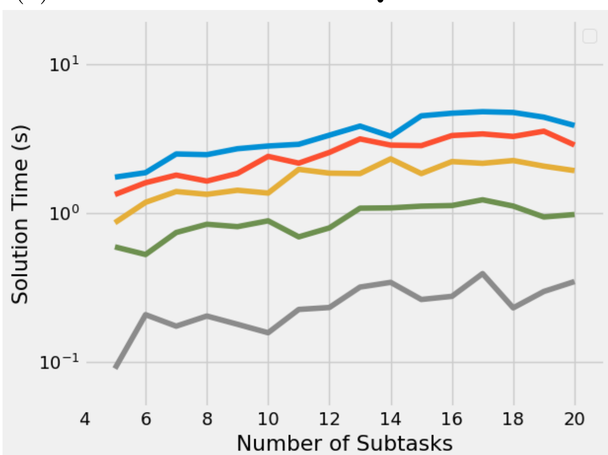
To more closely investigate the contributors to the execution time of the DP-based approach, we also decompose the execution time measurements, separately plotting the time to solve the multiple MIQPs for each task, and the subsequent time to solve the dynamic program. Results are plotted in Figure 6.12.



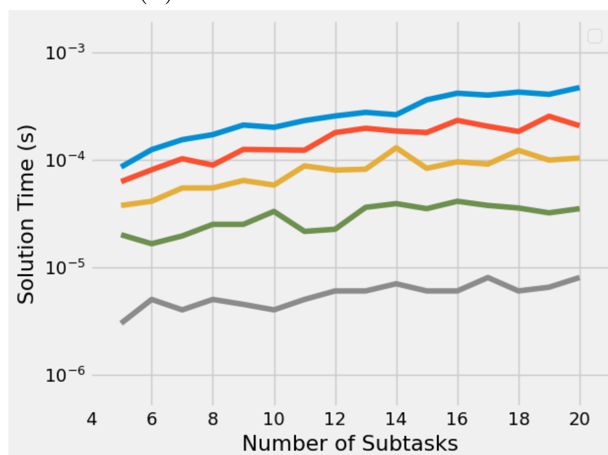
(a) Max Times to Solve MIQPs for Each Task



(b) Max Times to Solve DP



(c) Median Times to Solve MIQPs for Each Task



(d) Median Times to Solve DP

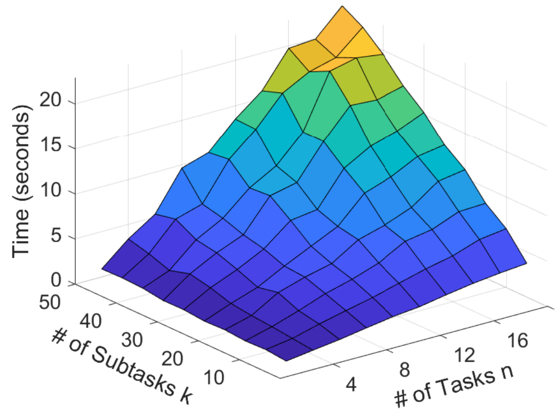
Figure 6.12: Contributors to execution time of the DP-based approach.

We observe that ***the time to solve the individual MIQPs for each task dominates***, while solving the DP itself is very efficient, taking only 6.8 ms in the worst case. Moreover, the median time to solve the DP increases by about 2 orders of magnitude from 2 to 10 tasks, which is what we expect from the theoretical upper bound of 125×. However, the maximum time to solve the DP for 2 tasks is 418 μs, only 16.4× faster than for 10 tasks, suggesting that for the tested tasks, the DP times scale slowly as tasks are added.

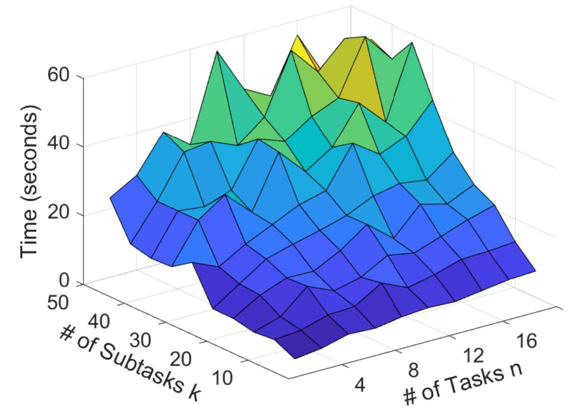
Evaluation of a Larger Parameter Space

We have shown that for even a large number of subtasks, solving an MIQP that represents span using a constraint for each edge for an individual task remains very efficient. We therefore expect our DP-based approach to remain feasible even for larger numbers of tasks with more complex control flows. Moreover, we expect that if the MIQPs are solved offline when task parameters are characterized, it will be possible to quickly solve the DP *online* to achieve real-time task adaptation.

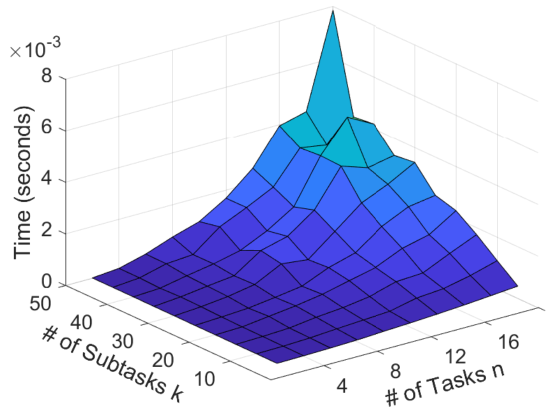
To test these hypotheses, we generate new task sets over a broader space of numbers of tasks and subtasks. This time, we use sets of size n from 2–20 in steps of 2, each assigned the same number k of subtasks from 5–50 in steps of 5. For each pair (n, k), we generate 100 task sets using the same methodology as before, again with an edge probability of $p = 0.05$. We split the work of applying the DP-based approach to these 10 000 task sets across 100 threads. Measured execution times are plotted in Figure 6.13.



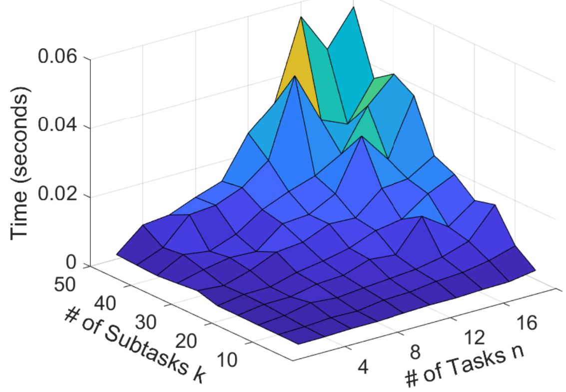
(a) Median Time to Solve MIQPs



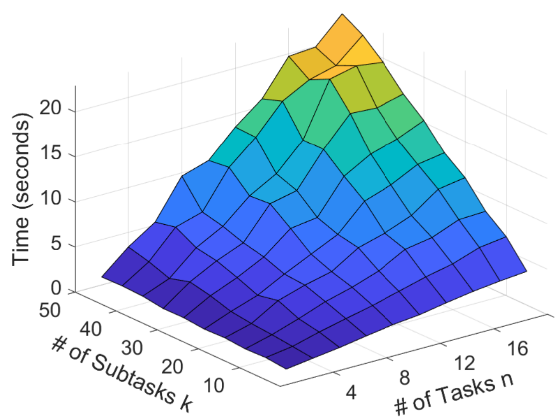
(b) Maximum Time to Solve MIQPs



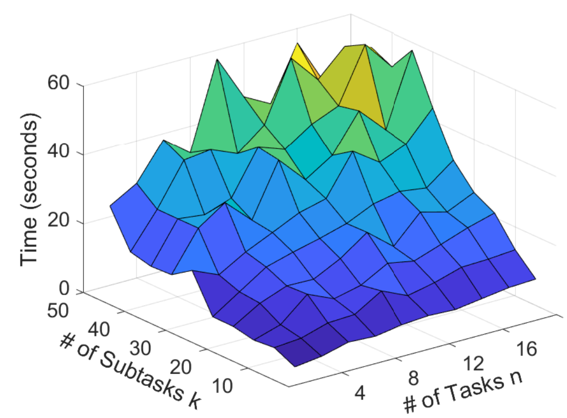
(c) Median Time to Solve Dynamic Program



(d) Maximum Time to Solve Dynamic Program



(e) Median Total Time



(f) Maximum Total Time

Figure 6.13: Execution Time Statistics for DP-Based Approach.

The results indicate that ***our DP-based approach remains efficient even for larger numbers of tasks*** — even for 20 tasks with 50 subtasks each, the total execution time remains less than one minute. Compared to solving a single joint MIQP, which can take over an hour for 10 tasks with 20 subtasks, this a substantial improvement. Moreover, ***just solving the DP is even faster***, taking under 60 ms for up to 20 tasks with 50 subtasks. Where the MIQPs can be solved offline, this can enable real-time online adaptation and re-allocation of cores, e.g., during admission control of new tasks, or when the number of available processors changes.

6.6.4 Comparison to Workload Compression in [119]

To complete our evaluation, we compare our model of subtask-level workload compression to the original approach to workload compression of parallel DAG tasks proposed by Orr et al. in [119]. We intend to characterize the extent to which, by considering that the span term L_i in Equation 6.2 decreases with the workload C_i, our model can reduce the amount of compression necessary to achieve schedulability, and in doing so reduce the minimum number of processor cores on which a task is still schedulable when compressed.

A direct comparison is difficult, because the original model also does not assign a unique elasticity to each subtask. We therefore construct our test as follows.

1. We use the randomly-generated DAG tasks from Section 6.6.2 that were used to evaluate the efficiency of each MIQP formulation to compress individual tasks. These consist of 1000 task sets for each number k of subtasks in 5–50.

2. For each task τ_i thus generated, we use the assigned values to compute the minimum and maximum total execution times $C_i^{\min}$, $C_i^{\max}$ and spans $L_i^{\min}$, $L_i^{\max}$.

3. We then use these values with Equation 6.2 to compute the maximum number of cores $m_i^{\max}$ needed to schedule task τ_i. We also calculate the minimum $m_i^{\min}$ to achieve schedulability under our model (using $L_i^{\min}$) and the minimum $m_i^{\min *}$ that arises from the model of Orr et al. due to keeping the span constant (using $L_i^{\max}$).

4. For each number of cores in $[m_i^{\min *}, m_i^{\max} - 1]$, we determine the workload C_i to achieve schedulability according to Equation 6.2 under the model of Orr et al., again keeping

the span fixed at $L_i^{\max}$. We compare this to the total workload $\sum_j c_{i,j}$ achieved by our MIQP-based model for those same numbers of cores.

Comparison of Minimum Core Allocations

We begin by comparing the values $m_i^{\min}$ achieved by our model — which is cognizant of the effect of minimum subtask workloads $c_{i,j}^{\min}$ on the minimum span $L_i^{\min}$ — to the values $m_i^{\min *}$ achieved by the model in [119] which holds span constant. Figure 6.14 shows the ratio $m_i^{\min}/m_i^{\min *}$ of the two values.

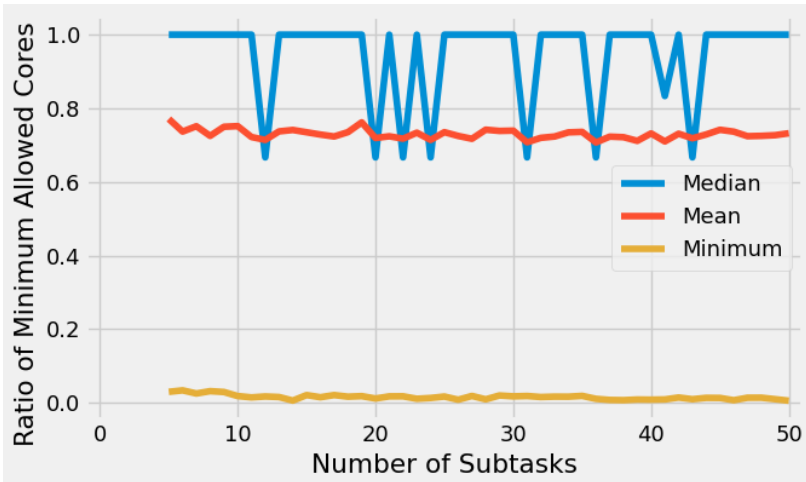


Figure 6.14: Ratio $\frac{m_i^{\min}}{m_i^{\min *}}$ of minimum allowed cores.

We observe that the ratio does not appear to be highly dependent on the number of subtasks. However, it does illustrate an important point: ***by also compressing task span, the minimum number of cores on which a task can be scheduled is decreased***. On average, the value $m_i^{\min}$ achieved by our subtask-aware model is $0.73\times$ the value $m_i^{\min *}$ achieved by the earlier model of Orr et al. And at best, $m_i^{\min} = 0.0057 \times m_i^{\min *}$. This implies that sets of tasks can be successfully compressed for schedulability on systems with fewer available cores. To better quantify this, we also measure the value

$$\frac{\sum_i m_i^{\min}}{\sum_i m_i^{\min *}} = 0.41$$

aggregating the minimum demand for cores under both models across all 46 000 sets of tasks. This suggests that, on average, our subtask-level elastic scheduling may be able to schedule sets of tasks on systems with 41% the number of cores needed by the method of Orr et al. in [119].

Comparison of Total Workloads

We next compare the amount of computational workload that remains available to tasks when compressed under each model. For every number of cores on which each task can be compressed by both models, we compare the values C_i achieved by our subtask-level elastic scheduling model, to the values C_i^* achieved by the earlier model of Orr et al. [119]. Figure 6.15 shows the ratio C_i/C_i^* of the two values.

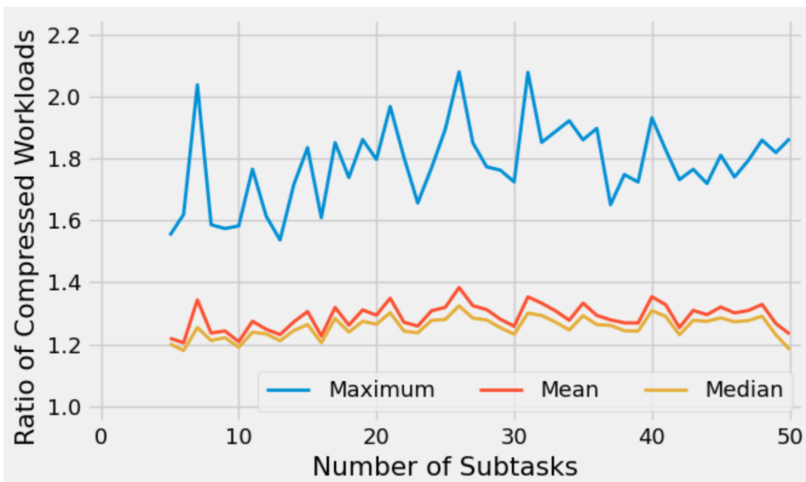


Figure 6.15: Ratio $\frac{C_i}{C_i^*}$ of compressed workloads.

Once again, we do not see a significant relationship between the ratio and the number of subtasks. Unsurprisingly, however, we do observe that ***by compressing task span, a task's total workload does not have to be compressed as much*** to be schedulable on a given number of cores. The median of the ratio C_i/C_i^* is 1.23, and at best, our subtask-level model achieves a workload 2.08× that of the original model in [119] among the tasks we tested.

6.7 Conclusion

In this chapter, we have presented a new model of subtask-level elasticity for federated scheduling of parallel tasks. The model considers the joint impact of compressing the workloads of *each* subtask within the task system, including changes to each task's span, which also affects the assignment of processor cores to each task.

We have illustrated two approaches to formulating the problem as an MIQP. Using an off-the-shelf solver, we have demonstrated that the problem can be efficiently solved to make offline scheduling decisions, even for complex task DAGs. We have also shown that, with proper offline characterization, a dynamic-programming (DP) algorithm enables pseudo-polynomial compression during online admission control. We have discussed how the same DP-based approach enables joint compression and dynamic core allocation to low-utilization tasks scheduled concurrently in a federated manner with high-utilization parallel tasks. We have outlined how to do this for both fluid and partitioned EDF scheduling of sequential tasks.

The results indicate that by solving a set of MIQPs offline to characterize discrete compression states, ***our subtask-level elastic scheduling model achieves millisecond-scale adaptation for online compression***. Furthermore, by compressing task *spans* in addition to their *workloads*, our model ***achieves smaller constraints on the minimum number of cores for each task, and does not have to compress a task's total workload as much*** to remain schedulable on a given number of cores, compared to the original model of Orr et al. [119] for workload compression in parallel DAG tasks.

Chapter 7

Parameterized Workload Adaptation for Fork-Join Tasks with Dynamic Workloads and Deadlines

Portions of this chapter were published as "Parameterized Workload Adaptation for Fork-Join Tasks with Dynamic Workloads and Deadlines" at RTCSA 2023 [146].

7.1 Introduction

In the previous chapters of this dissertation, we have explored the link between Buttazzo's elastic scheduling model [39, 40] and the optimization problem proposed by Chantem et al. [44, 45]. In particular, we have shown in Section 5.6.1 that this captures a quadratic first-order relationship between loss in system utility and the reduction of each elastic task's rate or workload. As we demonstrated for the real-time FIMS [166] and ORB-SLAM3 [42] applications, characterizing these relationships allows a system designer to assign quantitative and semantically-meaningful values to task elastic parameters. Furthermore, in Chapter 6, we proposed that for computationally-elastic parallel tasks, these parameters should be assigned to individual *subtasks*, rather than each task as a whole, to reflect the impact of each subtask's execution on system outcomes.

Nonetheless, these models remain limited in their ability to express richer semantics that may arise in the relationship between computation and subsequent result utility. Looking to the future, we propose that elastic scheduling should be reasoned about more broadly as an optimization-based framework in which tasks can be adapted to maximize result quality within the constraints of schedulability and other application-imposed constraints (e.g., safety requirements). Toward this vision, this chapter presents a new approach to workload compression for a single highly-parallel fork-join task executing on a fixed number of dedicated processors. It frames elasticity as a problem of adjusting a task's workload over multiple parameterized degrees of freedom with continuous or discrete values. By characterizing the impact of workload reduction on response time and utility, we generate a Pareto-optimal surface over which efficient search, interpolation, and extrapolation enable online selection of task parameters in response to dynamic factors, such as deadlines that are not known prior to job release. This is the first step toward the future goal of formalizing a complete optimization framework for execution of multiple tasks on a limited set of computational resources.

7.1.1 Contributions of This Chapter

Many real-time systems execute in dynamic environments where exogenous factors inform task workloads and latency requirements, which therefore might not be known prior to job release. If a job's workload cannot be completed in time, it nonetheless may be able to adjust its computation to provide an imprecise result prior to the deadline: anytime workloads [86] stop executing when their budget is exhausted, providing the current state of their results, while others support discrete execution modes corresponding to varying degrees of precision that can be selected prior to execution [121, 98, 140].

However, anytime or discrete semantics might not fully capture the dimensions over which a task's workload can adapt to meet its deadline. Some computations have *multiple* parameterized degrees of freedom that may be adjusted from their nominal values. These can be categorical (e.g., selecting from among a collection of algorithms) or numeric. Numeric parameters typically take discrete values (e.g., the number of iterations to refine a result), though at fine granularity, they can be approximated as a continuous state space (e.g., the proportion of input data selected from a large set for processing). If an instance of a task is not schedulable when run using its *desired* computational mode, its utilization may be reduced or *compressed* by adjusting these parameters to guarantee completion while minimally degrading result utility.

In this chapter, we consider the problem of parameterized workload adaptation for highly-parallel fork-join tasks with dynamic workloads and deadlines executing on a fixed number of dedicated cores. Given such a task, the challenge is to ① identify the parameterized degrees of freedom over which its workload can be adjusted to compress its utilization, then ② characterize the effect of compression on result utility. By also ③ quantifying the worst-case response time of the task as a function of those parameters on a given number of cores, we can formulate an optimization problem to select parameter values that minimize utility loss while constraining the response time according to the task's deadline. For example, as we discussed in Chapter 5, for simultaneous localization and mapping (SLAM) systems [88, 42], result utility can be scored quantitatively according to the relative translational error (RTE) [90] of a map; compression should therefore seek to maximize accuracy by minimizing RTE within the constraints of schedulability.

Several challenges must be addressed in the face of dynamic workloads and deadlines. Characterizing the objective as a closed-form function over multiple parameterized dimensions may be difficult, and finding optimal values for each parameter that satisfy the problem's dynamic constraints might be inefficient for online compression. Furthermore, while parameters must be assigned to satisfy schedulability under worst-case assumptions, avoiding unnecessary worst-case pessimism (i.e., overcompression) remains a goal of this chapter.

Our solution is to quantify loss empirically for a large set of states (i.e., joint parameter settings), constructing a monotonically-decreasing hull of hyperplanes between these states. The set of all states is reduced to a Pareto-optimal surface by sorting candidate states in order of worst-case response times for a target platform and removing those for which a *greater* response time yields a *lower* utility. At job release, this surface can be efficiently searched — and interpolated or extrapolated — to find a state satisfying the dynamic constraints imposed by the job-specific workload and deadline.

Selected parameter values are then applied according to application semantics, defining a computational mode prior to execution. Despite conservative parameter selection to guarantee schedulability under worst-case execution times, some applications enable less pessimism. For example, an execution time budget may be assigned to anytime subtasks to allow additional execution if assigned work is completed early, and alternative approaches of slack reclamation (such as we describe in Section 7.8) are sometimes possible.

We apply our techniques to the Advanced Particle-astrophysics Telescope (APT) [30], a planned orbital observatory (illustrated in Figure 7.1) that will detect and localize gamma-ray bursts (GRBs) in real time using onboard embedded hardware that is highly constrained in size, weight, and power (SWaP). By promptly communicating a GRB's direction to secondary instruments, APT will enable follow-up observations across a wide spectral range. We currently model GRB localization as a highly-parallel fork-join task with a workload and deadline that depend on the unique characteristics of each GRB [145]. We demonstrate that our approach enables efficient online compression and slack reclamation, allowing for rapid and accurate localization of even bright transient GRBs that may provide only a short window of opportunity for observation. On a quad-core ARM Cortex-A53 platform, we are able to localize 4 bright short GRBs to within a degree while meeting a 33 ms deadline, despite uncompressed localization requiring longer than a second.

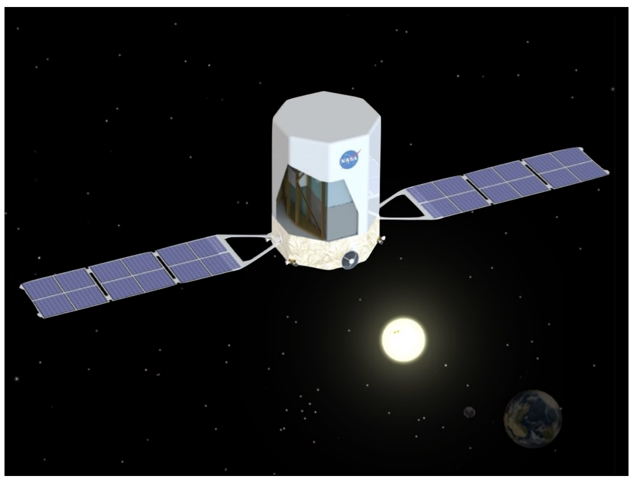


Figure 7.1: A rendering of the APT instrument [30].

7.1.2 Organization

The rest of this chapter is organized as follows:

- Section 7.2 reviews the elastic scheduling background relevant to this chapter and discusses some of the related work on other adaptive frameworks.
- Section 7.3 provides an explicit problem statement and describes the system model considered in this chapter.

- Section 7.4 outlines the solution approach that we propose.
- Section 7.5 describes the proposed APT satellite and its real-time GRB localization pipeline, which is the target application against which we apply and evaluate the techniques proposed in this chapter.
- Section 7.6 identifies the parameterized degrees of freedom over which the GRB localization workload can be adjusted, and characterizes their impact on loss of result utility — which in this case is defined as the error (in degrees) of the localized source direction inferred for the burst.
- Section 7.7 quantifies the impact of each parameter on the localization task's workload and response time, deriving a closed-form function for each phase of the task.
- Section 7.8 presents our efficient implementation of our adaptive framework for APT and discusses additional optimizations, such as a method to reclaim slack time if localization finishes earlier than predicted by its worst-case response time.
- Section 7.9 evaluates our adaptable GRB localization pipeline in the context of synthetic GRB data using representative spectral characteristics, as well as simulations of four catalogued real-world GRBs.
- Section 7.10 concludes the chapter and suggests how the work is situated within our broader vision.

7.2 Background and Related Work

The elastic scheduling models discussed in this dissertation — including the prior models of Buttazzo et al. [39, 40], Chantem et al. [44, 45], and Orr et al. [119, 120, 118, 121], as well as those new to this dissertation — already offer a collection of frameworks for dynamic adaptation of task utilizations to avoid system overload. Though several of these models support computational elasticity — i.e., compressing task workloads — they do not describe *how* to adapt computation under the applied constraints. The discrete elastic model of Orr et al. [121] considers specific modes of execution, but this does not capture the multiple degrees of freedom (which may be continuous, discrete, or categorical) over which a task's workload may be compressed. The AutoE2E adaptive framework for autonomous vehicles [8, 7] supports subtask-specific objective functions, but it is limited to end-to-end sequential execution and does not consider how compressing a subtask may affect its successors.

In [128], the authors survey protocols for switching between execution modes without missing deadlines. An adaptive framework in [27] degrades task execution according to "service levels," with each assigned an "importance value" reflecting a user-defined notion of quality of outcome. While such protocols may help to inform the transition between uncompressed and compressed execution, we propose a richer elastic model to support tasks having both discrete and continuous modes of operation, as well as combinations of the two.

A similar approach was presented for parallel LiDAR object detection [140] and applied to the PointPillars [87] encoder, aiming to maximize utility while maintaining schedulability guarantees by profiling execution times and assigning accuracies to discrete states. Such approaches may allow for a more objective measure of performance than traditional elastic scheduling, which compresses task utilizations proportionally to their elasticity. However, they are limited in practice to a small set of discrete states. We propose to allow adaptation according to a user-defined objective function over multiple degrees of freedom involving continuous or discrete numeric values and categorical variables.

A more general framework for allocating limited resources (e.g., CPU resources to guarantee schedulability) while maximizing application utility was provided by Rajkumar et al. [126] in their resource allocation model for quality-of-service (QoS) management. Similarly to the work of this chapter, this allocation model treats utility as a function of multiple parameters, and these functions are restricted to be monotonically increasing in each resource dimension. Each parameter represents some system resource; the joint allocation of each resource to each application must not exceed the total availability of that resource on the system. CPU bandwidth/utilization may represent one resource, but this provides an abstraction model over other types of computational resources. In their later work, Rajkumar et al. [127] propose techniques to treat the problem as one of optimizing over a convex hull representing the utility function; the solution approach in this chapter is similar. However, these models are fundamentally different from the model of this chapter: while we also consider utility as a function of multiple parameters, our parameters do not represent limited resources that must be allocated. Instead, *schedulability based on response time as a function of those parameters*, as opposed to *a total bound on the sum of each parameter*, is the first-class concern that forms the constraint under which we seek to optimize utility.

Recent work on dynamic deadline-driven execution [68] presents a novel adaptation scheme for tasks with environment-dependent deadlines and execution times. This data-driven execution framework provides handlers for deadline misses, allowing downstream components in the computational pipeline to adapt in response. The authors argue that periodic execution models, which use conservative WCET estimates, fail "to maximize the runtime-accuracy trade-off due to the large skew between the mean and maximum runtime," which typically "leaves plenty of slack." In Section 7.9, we demonstrate an approach that reclaims slack to provide higher utility while still using conservative WCET estimates to prevent deadline misses.

Our work in this chapter focuses on highly-parallel fork-join tasks, for which it is straightforward to characterize a closed-form worst-case response time under a nearly optimal schedule as a function of its compressible parameters. Many real-world applications [141, 168], including the GRB localization pipeline that we have developed for the proposed APT mission [30, 48, 144, 153, 171, 145, 147, 75, 47, 146, 76, 154] considered in this chapter, can be described as such.

7.3 System Model and Problem Statement

This chapter focuses on recurrent, constrained-deadline, *highly-parallel fork-join* tasks. Though both chapters deal with parallel tasks, the task model in this chapter differs fundamentally from the parallel DAG task model described in Section 6.2.

In this chapter, every job $J_{i,k}$ of a task τ_i is characterized by a relative deadline $D_{i,k}$. This parameter encodes the same meaning as for the task models in prior chapters, representing the interval after the job's release by which it must complete execution. However, unlike in prior chapters, the deadline $D_{i,k}$ may be unique to each job, and might not be known until the release of the job. Deadlines are still constrained to not exceed the task period — i.e., each job must complete prior to the release of the next job. Using gamma-ray burst (GRB) localization as an example, the release of an instance of the localization task, corresponding to the detection of a GRB, has a deadline that may depend on several factors, including the set of instruments available for follow-up observations and their associated communication latencies and slewing speeds [145]. Because these are infrequent enough events, we may assume that the deadline will have passed prior to the detection of another GRB.

Similarly, each job has a workload C_i that may not be known prior to its release. Again, the parameter has an equivalent semantic meaning to the workload of previous chapters, representing the worst-case execution time of the job if executed on a single processor core. But, since workload depends on external factors (e.g., the amount of data, corresponding to the number of gamma rays that interact with the detector for a given burst), we cannot make response-time guarantees without adapting task execution for a given job.

Unlike under the DAG task model, this chapter treats highly-parallel fork-join tasks as a sequence or chain of subtasks $\{\tau_{i,j}\}$ with workloads $C_{i,j}$, where each subtask is either *sequential* (s) or *parallel* (p), as illustrated in Figure 7.2. Unlike the parallel DAG task model in Chapter 6, this means that a single subtask is not restricted to executing sequentially. Under the model considered in this chapter, a parallel subtask can divide its workload evenly across processor cores, similarly to the gang and bundled task models described in [170].[22]

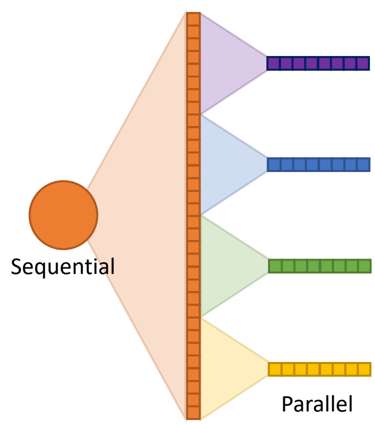


Figure 7.2: A highly-parallel fork-join task with a sequential subtask followed by a parallel subtask.

Unlike the gang or bundled task models, this chapter restricts each task to execute on a fixed number of *dedicated* cores m_i, similarly to the federated scheduling model of [94] discussed in Chapter 6. Since the model under consideration allows parallel subtasks to distribute

[22]While such a representation is limited to a more restricted class of parallel control flow compared to the general DAG task model, it provides additional flexibility to parameterize workloads across parallel regions. As we will see, our parameterized workload model applies to parallel subtasks, enabling execution times to be shortened by, e.g., reducing the amount of data to be processed in parallel. To represent this under parallel DAG task model, a parallel subtask would have to be decomposed into a large number of subtasks, each representing a single unit of sequential execution (e.g., a single data item). Equivalent parameterization would have to reduce the number of *subtasks*, not just the workload of individual subtasks; these semantics are not captured by the elastic models for parallel DAG tasks that we have discussed so far in this dissertation.

their workloads evenly across processors, the worst-case response time $\mathcal{R}_i$ of task τ_i can be expressed as

$$\mathcal{R}_i = \sum_{\tau_i \text{ is } s} C_{i,j} + \sum_{\tau_i \text{ is } p} \frac{C_{i,j}}{m_i} \tag{7.1}$$

This implies that the task is schedulable if and only if $\mathcal{R}_i \leq D_{i,k}$ for each job $J_{i,k}$; offline schedulability analysis is therefore challenging, since deadlines are not known a priori. In this chapter, we address the scenario where a job is released with a workload and deadline for which it is not schedulable. We consider computationally elastic tasks having multiple execution states associated with a set of application-specific parameters $\{a_\ell\}$ over which workload can be adjusted according to one of the following semantics, allowing response time to be expressed as a monotone non-decreasing function $\mathcal{R}(\{a_\ell\})$:

1. The workload of one or more subtasks may be a function $C_i(\{a_\ell\})$ of discrete or continuous numeric parameters; these must be monotone non-decreasing in each parameter. Section 7.6 provides examples of subtasks having execution times linear or quadratic in a continuous numeric parameter representing the amount of input data to process.

2. A discrete numeric parameter (e.g., a number of iterations) may change the sequence of subtasks. In this work, we consider the case where such a parameter defines the number of identical copies of a sequence of subtasks.

3. A categorical parameter may change the computational mode (e.g., the algorithm used) of one or more subtasks. In this case, each mode may impose its own workload as a function $C_i(\{a_\ell\})$ of the other parameters. To match the semantics of (1), we assign numeric values to each category, with the requirement that a larger numeric value is assigned to a mode with a greater workload.

Each parameter a_ℓ is constrained by some maximum value $a_\ell^{\max}$. These values may be either constants or monotone non-decreasing functions of another parameter. The *uncompressed* workload is defined as that associated with each parameter taking its maximum value. Formally, a task with a dynamic workload is one for which some values $a_\ell^{\max}$ are unknown prior to job release. If a job is not schedulable, its workload is *compressed* by selecting values $\{a_\ell\}$ such that its response time does not exceed its deadline. Compression should attempt to maximize the utility of the system by minimizing some application-specific loss function $\mathcal{L}(\{a_\ell\})$ of these parameters. We consider applications for which $\mathcal{L}$ is monotone non-increasing with each parameter, i.e., doing *more* work yields a better result. We formulate our problem as follows:

$$\min_{\{a_\ell\}} \quad \mathcal{L}(\{a_\ell\}) \tag{7.2a}$$

$$\text{s.t.} \quad \mathcal{R}(\{a_\ell\}) \leq D \tag{7.2b}$$

$$\forall_j,\ a_\ell^{min} \leq a_\ell \leq a_\ell^{max} \tag{7.2c}$$

Here, a_ℓ^{min} constrains the parameter to some minimum value and is assumed to be known a priori. The problem, then, is to identify the parameters over which a task's workload may be compressed; to characterize their impact on the task's response time and the utility of its result; and finally to use this information to solve optimization problem in Expression 7.2 efficiently online to adjust a released job's computation to adapt to overload.

7.4 Solution Overview

This section provides our solution approach to the problem posed in the prior section for highly-parallel fork-join tasks executing on a fixed number of dedicated cores. In particular, we demonstrate how ***offline workload parameterization enables construction of a Pareto-optimal surface that can be searched online to adapt task execution in bounded time.*** In subsequent sections, we illustrate and evaluate our approach in the context of a GRB localization for APT.

7.4.1 Offline Steps

Parameterizing a task's workloads, then characterizing the joint dependence of response time and result utility on those parameters, is performed offline. The resulting Pareto-optimal surface can then be searched online in time polynomial in its representation, enabling real-time task adaptation in the face of dynamic workloads and deadlines.

Step ①: Identify Parameters

Parameters may be identified offline by inspection of the application and should match the semantics listed in Section 7.3. The parameters for adapting APT's GRB localization task are described in Section 7.6.

Step ②: Characterize an Objective Function

Utility loss for an application can be quantified empirically for a large set of input state combinations. For each parameter a_ℓ, a number b_ℓ of values within the state space should be considered. The complete Cartesian product of these values should be tested, except where some parameter is constrained by another. Smaller values of b_ℓ reduce the number of samples for efficiency of offline analysis, but denser sampling may allow for more accurate characterization of the objective (and the input space may still be reduced in Step ④ for use online). The selection is left up to the application designer, though for categorical parameters, each possible value should be considered. In Section 7.6, we identify 4 parameters and test 2657 input states for GRB localization.

For x numeric and y categorical parameters, the objective will be a function of $x + y$ dimensions, characterized as $\prod_y b_\ell$ x-dimensional manifolds. Fitting a closed-form function to the losses observed at the sampled states may be difficult and error-prone. Instead, our approach allows the loss function to be represented as a monotonically decreasing hull formed by hyperplanes connecting the space of observed states. Construction from a Pareto-optimal subset of states is described in Step ④.

Step ③: Quantify Response Time

The task's worst-case response time can be quantified by decomposing it into constituent subtasks, then profiling subtask execution times individually or in groups that share dependence on common parameters. Execution times $C_{i,j}(\{a_\ell\})$ as functions of the input parameters can thus be characterized for those parameters satisfying semantic (1) from Section 7.3. For those parameters a_ℓ satisfying semantic (2), some subset of subtasks is duplicated by the parameter value, equivalent to scaling the workload of the individual subtasks by a_ℓ. This can be incorporated into the expression $C_i(\{a_\ell\})$. From this, Equation 7.1 can be used to compute response times as functions of execution times, allowing easy adjustments of core assignments for the application on a given platform. For categorical parameters, the response-time functions for the Cartesian product of their values must be identified, resulting in up to $\prod_y b_\ell$ functions of the form

$$\mathcal{R}_i(\{a_\ell\}) = \sum_{\tau_i \text{ is } s} C_i(\{a_\ell\}) + \sum_{\tau_i \text{ is } p} \frac{C_i(\{a_\ell\})}{m_i}. \tag{7.3}$$

While the set of functions grows exponentially in the number of categorical parameters, realistically this can be represented more compactly. Such parameters typically represent the selection between a handful of available computational modes or algorithms to apply to a phase of the application. These are selected by the application designer and so may be as small in number as desired. For example, in Section 7.6, a single categorical parameter selects one of two algorithms for an initial approximation of a gamma-ray burst's direction; this selection only affects the response time of the approximation stage subtasks. As the results in Section 7.9 demonstrate, a single such parameter is sufficient for our GRB localization task.

Step ④: Generate Pareto-Optimal Surface

Candidate states are sorted by response time, after which any state with a higher loss than the previous state (i.e., for which a higher response time results in a worse outcome) is discarded, leaving a set $\boldsymbol{S}$ of states ξ. As noted in Section 7.8, for GRB localization aboard APT, this procedure yields fewer than 100 candidate states for each platform we tested. From these, we construct hyperplanes connecting adjacent states for interpolation along the Pareto-optimal surface. For each candidate state ξ, we find the points from the *original* set of states having the next larger value of each parameter respectively with lower error,[23] holding constant the other parameters in ξ. These hyperplanes can be extended for extrapolation to parameter values beyond the ranges used to infer the surface.

[23] Due to the often stochastic nature of characterizing loss, some adjacent states may not have a lower objective value for a larger parameter value.

7.4.2 Online Steps

To implement online task compression, we modify the task to include an initial sequential subtask that calculates its response time according to the revealed constraints on workload parameters at time of release. In an overload scenario, the subtask should then solve the optimization problem in Expression 7.2 and apply the resulting parameters. Realizing computational mode changes is application-specific, but we outline an OpenMP-based approach for GRB localization in Section 7.8. As this subtask adds to task workload, it must remain efficient and be accounted for in the response time.

Step ⑤: Check for Overload

When a job of a dynamic task arrives, the initial subtask must determine if the job will complete in time. To do so, it calculates response time based on the parameter constraints revealed. We assume that each function $C_i(\{a_\ell\})$ can be computed in time linear in the number of numeric parameters x, so for a given input state, response time can be calculated in time $\mathcal{O}(x_d m)$ for m subtasks and x_d numeric parameters with dynamic constraints.

Step ⑥: Online Solution Search

If a job's response time exceeds its deadline, the Pareto-optimal surface can be searched in time $\boldsymbol{\mathcal{O}(\log|S| + x)}$ for a set of parameters that satisfy schedulability. Binary search ($\mathcal{O}(\log|S|)$) over the sorted set of candidates finds the state ξ with the greatest response time not exceeding the deadline, from which a Pareto-optimal solution is then obtained by interpolation or extrapolation. This can be performed efficiently by considering each parameter in ξ connected to an adjacent state in Step ④, with the other parameters held constant, solving in constant time for the value yielding a response time equal to the deadline. The best such value obtained (i.e., the one corresponding to the state with the lowest objective function value) is chosen. This takes total time linear in the number of numeric parameters x.

In the case that the state ξ has values that exceed the dynamic parameter constraints imposed on the job, iterative search down from ξ can be used to find the best state ξ' for which all parameters are within the constraints. However, parameter extrapolation from this state is not guaranteed to find a Pareto-optimal set of parameters, though the values will have a higher expected utility than ξ'. This is not a problem for our target application, as its dynamic constraints (described in Section 7.6) are defined by the amount of input data available, and our real-world test cases (described in Section 7.9) all provide sufficient input data. As such, exploration of alternative approaches (such as storing multiple surfaces, or falling back to iterative search over the complete set of candidate states generated in Step ③) are deferred to future work.

Step ⑦: Adapt Task Execution

Once compressed parameters are found, a task may execute in its degraded state. For subtasks with discrete modes, execution should proceed according to the state defined by the input parameters, but this may result in overcompression as worst-case response times are often pessimistic. For collections of subtasks with anytime workloads, workload compression can instead be applied by calculating WCETs corresponding to the given input parameters. This portion of the task may then be allowed to proceed until the compressed WCET or response-time limit has been reached, whereupon it is stopped and the current result is used. Some applications may provide other opportunities to reduce pessimistic overcompression via slack reclamation; we describe one such approach for GRB localization in Section 7.8.3.

7.5 Target Application: GRB Localization

The National Academies released the Astro2020 decadal survey [117] to identify scientific challenges for astronomy and astrophysics in the next decade. It highlighted the "space-based time-domain and multi-messenger program" as the highest-priority sustaining activity in space, which will require coordinated real-time follow-up observations of transient astrophysical phenomena – e.g., gamma-ray bursts (GRBs) — using secondary observational modalities, such as visible-light observations with optical telescopes.

Pursuant to this, the Advanced Particle-astrophysics Telescope (APT) [30] is a planned space-based observatory[24] that aims to further scientific understanding of the nature of dark matter and the physics of neutron-star mergers by supporting multi-wavelength and multi-messenger astrophysics [11, 114, 106] through rapid detection of gamma-ray bursts (GRBs). It will subsequently direct secondary follow-up instruments to observe GRBs across broad ranges of wavelengths and emission modalities. APT will be deployed in a Sun-Earth Lagrange L_2 orbit, where obscuration of the sky by the earth is minimized and the benefit of its large (nearly full-sky, 4π-steradian) field of view can be exploited [30]. However, many optical follow-up instruments have narrow apertures (often $<1°$) and so must point almost directly at the GRB source. Because GRBs are transient events, long delays from initial detection of a GRB's light to ground-based computation of its location in the sky (which is nontrivial to infer from the incoming gamma rays but is necessary to physically aim the follow-up instruments) cause lost opportunities for observation. APT will therefore perform *onboard* detection and localization of GRBs *in real-time*, enabling prompt communication of precise source directions to those secondary instruments.

APT's primary detector instrument will have 20 layers of sodium-doped cesium iodide (CsI:Na) scintillating crystal tiles. From a single burst, thousands to millions of gamma-ray photons are expected to enter the detector. Each gamma ray may Compton scatter (illustrated in Figure 7.3) one or more times before being photoabsorbed; each such interaction is referred to as a *hit*, and a single photon's hits are collectively referred to as an *event*.

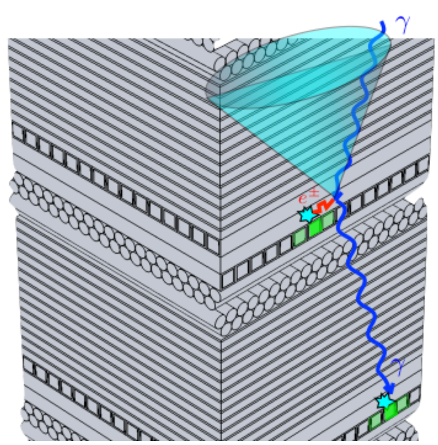


Figure 7.3: A 2-hit event in APT, with a single Compton scatter then photoabsorption.

[24] A smaller Antarctic Demonstration instrument, ADAPT, is scheduled to make a high-altitude balloon flight over Antarctica during the 2025–2026 season [147, 75, 47, 76, 154].

Optical photons produced by the energy deposits in the crystals due to these hits are captured by perpendicular arrays of wavelength-shifting optical fibers running across the top and bottom surfaces of each tile. Each fiber is read by a photodetector coupled with an analog-pipeline waveform digitizer ASIC [21]. The output of each array is processed by a single FPGA (e.g., a rad-hard Microchip RT PolarFire), which performs data pre-processing and reduction steps including pedestal subtraction, time-integration of signal intensities over a $\sim 2\mu$s sampling window, then zero suppression [154]. The FPGA then demarcates the boundaries of islands of contiguous positive signal values in the array; these regions are centroided by a second FPGA to infer interaction positions and energies for each hit.

Centroids are sent over Gigabit Ethernet to a back-end analysis CPU that aggregates and combines the received data packets to build the hit data corresponding to each event. This event building stage is still under development; we assume that once 95% of a burst's incident events have been captured and built, associated hits are available all at once in main memory, at which point an instance of the localization task is released.

The GRB localization task forms the highly-parallel fork-join computation illustrated in Figure 7.4. The task can be decomposed into a sequence of subtasks that collectively form the three stages detailed in Section 7.6: it reconstructs the gamma ray's trajectory through the detector, inferring the hit order and associated uncertainty according to the algorithms in [144, 171]. It then combines data from multiple reconstructed photons to approximate then refine an inferred source direction. Localization must be completed in time to guarantee prompt communication with secondary instruments while executing on a fixed number of cores in SWaP-constrained hardware flying aboard the orbital platform.

Each instance of this task, corresponding to the detection and localization of a unique GRB, is highly variable in its workload and deadline. The workload depends on the number of detected events, which itself is a function of the burst's spectral-energy distribution, angle with respect to the detector, and fluence.[25] The deadline may depend on the burst duration, which can range from around 10 ms to 20 minutes in the Compton regime [71, 162, 23, 163]. It also may be informed by the communication latency and slewing speeds of available follow-up instruments. Speed-of-light delays to ground-based devices impose an extra $\sim$5 s of latency, but APT may also be coupled with an onboard optical telescope (similarly to

[25]Fluence is a measure of total energy per unit area; for a given spectrum and source vector, fluence is proportional to the number of incident gamma rays.

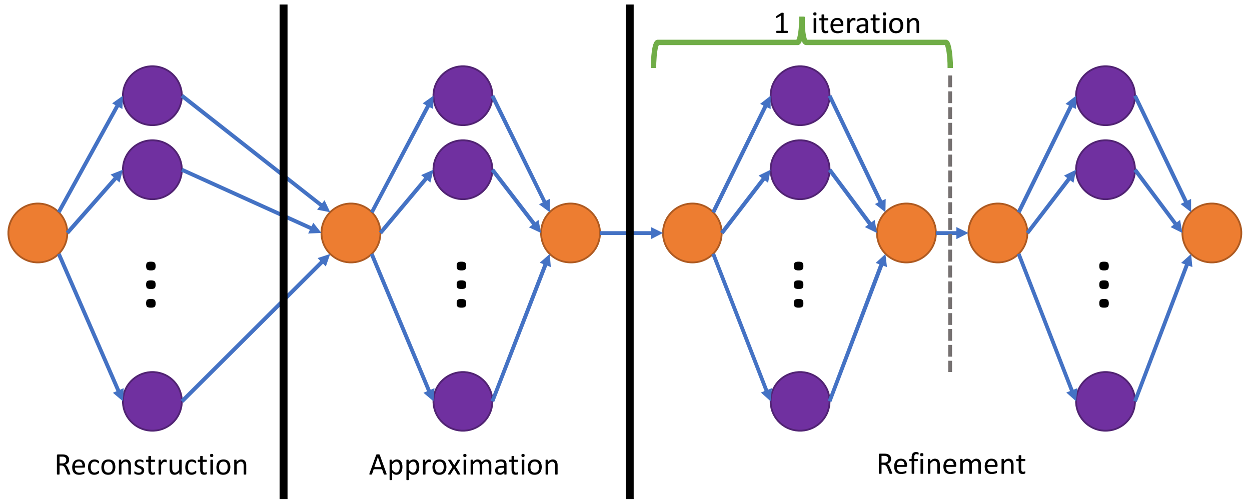


Figure 7.4: APT's highly-parallel fork-join GRB localization task.

Swift's UVOT [131]) with minimal communication latency. The deadline will ultimately depend on the window of time after the initial burst during which follow-up observations are useful; for short bursts (<1 s), the window of opportunity for follow-up observations may be very brief, though the timescale of prompt emissions in secondary modalities is still an open question in astrophysics that APT aims to address. The localization task therefore must be adaptable to guarantee completion before a deadline that may not be known a priori, even for highly transient bursts generating large volumes of data.

To characterize system performance, we simulate the APT instrument's response to several GRBs with our own implementation of the APTSoft package [48], rewritten in C++ for more efficient simulation. It uses the Geant4 simulator [3] to generate independent gamma rays from a simulated source and track their physical interactions in the detector. APTSoft subsequently models the resulting scintillations, light transmission through the optical fibers, waveform digitizers, and time integration. With our own software, we then centroid the simulated outputs and combine detected hits.

We generate sets of 10^6 gamma rays using two spectral-energy distributions characteristic of short GRBs [113]. We use two Band [10] functions with parameters α=−0.5, E_{peak}=490 keV, $\beta \in \{-3.2, -2.1\}$ to capture a range of spectral profiles. Spectral energies are in [100 keV−30 MeV] to match the Compton regime of the Fermi Gamma-ray Burst Monitor (GBM) sensitivity [122], data from which the distributions presented in [113] were obtained. For each spectrum, we first generate a normally-incident set, and then the sets described by the

Cartesian product of $\{30°, 60°\}$ polar angles and $\{0°, 45°\}$ azimuth angles. This gives a total of 10 synthetic GRBs across 5 incident angles and 2 spectral energy distributions, which we use to characterize the pipeline's localization accuracy and worst-case execution times.

7.6 Parameters and Loss Function

Each of the three stages of APT's GRB localization pipeline (illustrated in Figure 7.4) that execute on the CPU are amenable to workload compression, adapting to a dynamic deadline known only when each job is released — i.e., when a GRB is detected. Compression aims to minimize an objective function informed by the angular error in the predicted GRB source direction, while still guaranteeing that the deadline is met. The associated compressible parameters are outlined in Table 7.1.

Param	**Stage**	**Description**	**Constraint**
n_r	Reconstruction	Events to reconstruct	$30 \le n_r \le n_e$
α	Approximation	Approximation technique	$\alpha \in \{\textbf{FibSpiral}, \textbf{ApproxCircles}\}$
n_s	Approximation	Number of rings to sample for joint log-likelihood	$\max\{10, n_a\} \le n_s \le \min\{1000, n_a\}$
x	Refinement	Refinement iterations	$x \in \{0 \ldots 20\}$

Table 7.1: Compressible parameters for APT's GRB localization task.

7.6.1 Stage 1: Event Reconstruction

The timescales at which a single gamma-ray photon interacts within the detector are too short to directly determine an ordering. Instead, for each event, we use our tree search algorithm from [144] to infer the temporal order of the first two hits, constraining the gamma ray's source vector to the circle illustrated in Figure 7.3. For events with only two hits, we reconstruct the circle for both orderings. In simulation, events with more than 6 hits are extremely uncommon (<0.01%), so we exclude these from reconstruction to bound the size of the tree.

Uncertainty in detector spatial and energy measurements "smears" each circle into an annulus, often referred to as a "Compton ring" [28]. For events with at least 3 hits, we infer the single most likely ordering. Likelihoods cannot be assigned to the two possible orderings for events with only 2 hits, so we reconstruct and propagate a ring for both orderings. Each annulus is characterized by its direction, opening angle θ, and thickness $\delta(\cos\theta)$. Physically impossible reconstructions (those for which the Compton law would imply an impossible opening angle ($|\cos\theta|>1$) are dropped, and the remaining n_a annuli are passed to localization.

In the event of overload, reconstruction can degrade by dropping events. For n_e reconstructable events, we can reduce n_r, the number of events to be reconstructed, to a value $n_r < n_e$. As n_e is typically on the order of several thousand or more, we approximate n_r as continuous. Reconstruction is an anytime workload: the stage can stop at any point (e.g., when n_r events have been reconstructed) at which time any reconstructed annuli are passed to the next stage. Reconstruction processes events in order of arrival; compressing n_r defines a stopping point for the stage. We defer a more complex model that separately considers reconstruction of 2-hit and $\geq$ 3-hit events to future work.

To characterize the impact of compressing n_r, we iterated over a geometric progression of 11 values from 30 to 30 000, using uncompressed values for all other input parameters. For each value of n_r, we generated 10 000 inputs to the pipeline by randomly sampling 1000 subsets of reconstructable events from each of the 10 simulated GRBs. Figure 7.5 plots the discrepancy in degrees between the inferred and true source direction against the number of events reconstructed, with the vertical bars enclosing the extent of the distribution. Because of the high variance in localization error for a given value of n_r, rather than using expected error as the objective, we instead use 68% containment (representing the 68$^{\text{th}}$ percentile error — roughly 1σ if errors are Gaussian — a commonly used metric for GRB localization accuracy [48, 50]). These values are also shown in Figure 7.5, which illustrates a roughly log-log linear dependence on n_r.

7.6.2 Stage 2: Initial Source Approximation

We use multilateration over reconstructed annuli to infer the GRB's source direction. This involves an initial rough approximation that is then iteratively refined in Stage 3. We consider two approximation techniques (α) both of which execute over a subset $n_s \leq n_a$ of the input annuli. Our prior work [144, 171] fixed $n_s = 1000$ and used only the first approximation technique (**ApproxCircles**). It uniformly distributes 720 points around each of 20 circles

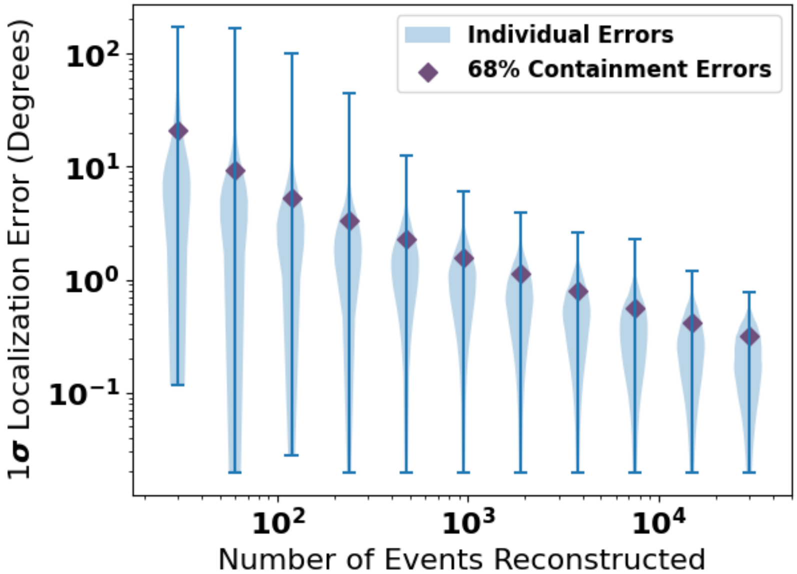


Figure 7.5: Impact of n_r on localization error. Note that axes are logarithmic.

selected at random from n_s, finds the point from each with the greatest joint log-likelihood over all n_s annuli, then uses a weighted mean to approximate the source vector. The second technique (**FibSpiral**) is new to the work in this chapter. It generates 100 points almost uniformly over the surface of the unit sphere with a Fibonacci spiral. For each point, it finds the joint log-likelihood over all n_s annuli, then approximates the source vector as a weighted mean over the top 10. Approximation requires both parameters α and n_s to be specified prior to computation.

While **FibSpiral** is much faster (requiring only $100{\cdot}n_s$ log-likelihood computations, versus $14\,000{\cdot}n_s$ for **ApproxCircles**), it has less fidelity in its estimate for equal values of n_s. In this work, we constrain n_s to the range $[10, 1000]$, which with the choice of α approximates two continuous state spaces that are non-overlapping in execution time but may overlap in result accuracy, as illustrated in Figure 7.6. Measured 68% containments for the approximated source error (degrees) without refinement are plotted against the number of log-likelihood computations required by values of n_s for each technique. For this plot, no subsequent refinement is performed, and n_r is fixed at 30 000. 68% containments for each n_s were obtained from 1000 trials over each simulated GRB.

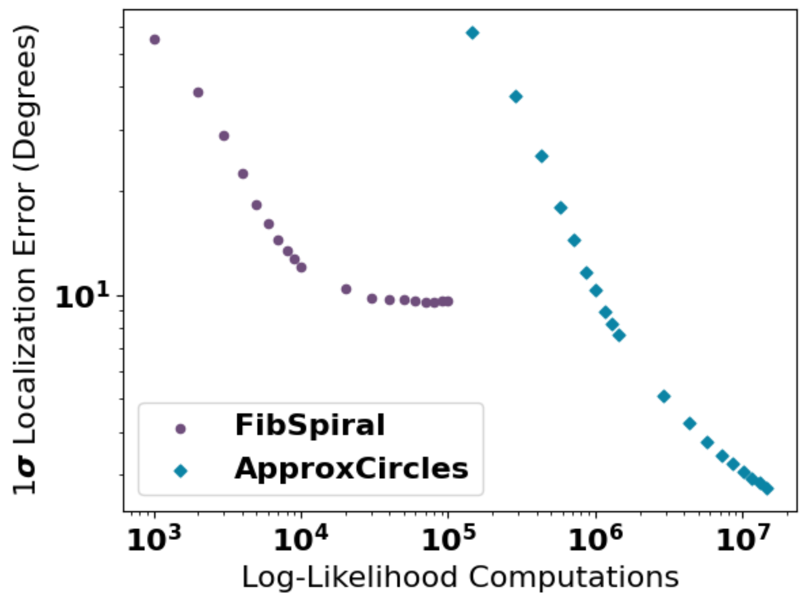


Figure 7.6: Comparison of approximation techniques.

7.6.3 Stage 3: Iterative Source Refinement

The approximation result is subsequently refined using a modified version of the iterative linear least-squares approach in [144, 171] over all reconstructed data. Refinement executes for x iterations (or until convergence). Whereas our prior work fixed x=20, now we allow the task to adapt by compressing x to a discrete numeric value in the range $\{0...20\}$. Iterative refinement can be terminated at any time, with the result of the last completed iteration (or the initial approximation, if no iterations completed) used as the estimated source direction of the GRB.

Iterative refinement is highly dependent on the quality of the initial source estimate provided by approximation, as illustrated in Figure 7.7. Each value of (n_s, x) is plotted against the 68% containment of localization error (degrees) over 1000 trials from our 10 synthetic GRBs with $n_r = 1893$ (the smallest value in our geometric progression for which n_s can reach 1000) and using the **ApproxCircles** technique. With fewer annuli sampled for approximation, more refinement iterations are necessary to converge on an accurate result. With more refinement iterations, the impact of a poor initial estimate is reduced.

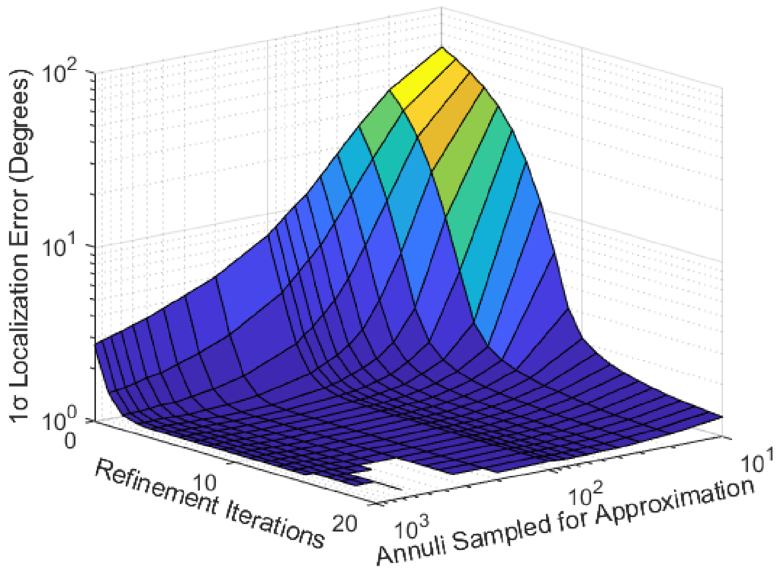


Figure 7.7: Impact of approximation on iterative refinement.

7.7 Response Times

As described in Section 7.6, generating a Pareto-optimal convex hull that can be efficiently searched to achieve elastic workload compression requires response times to be characterized as functions of the input parameters. Each of our GRB localization task's three stages of CPU computation (illustrated in Figure 6.1) has an initialization subtask preceding a parallel subtask, with the work evenly split across cores using OpenMP. We measure execution times for each subtask on the three quad-core hardware platforms listed in Table 7.2. These platforms span a range of performance in available SWaP-constrained hardware while remaining tolerant to harsh environmental conditions. Though not rad-hardened, they are candidates to fly aboard the scheduled Antarctic high-altitude balloon demonstration mission (ADAPT), and the Atom-based platform has already been selected for use as ADAPT's flight controller.

We quantify execution time performance by reading from the standard library's high resolution clock (`std::chrono::high_resolution_clock`). Compilation is performed using the Gnu Compiler Collection (GCC) at optimization level `O3`. To avoid interference from other processes, we run the pipeline at the highest priority under Linux's `SCHED_FIFO` scheduling class and disable CPU throttling.

Platform	Abbr.	CPU	Freq.	RAM	Linux v.
Raspberry Pi 3 Model B+	RPi3	Cortex-A53 (ARMv8)	700MHz*	1GB	5.15.61
Raspberry Pi 4 Model B	RPi4	Cortex-A72 (ARMv8)	600MHz*	4GB	5.15.61
WINSYSTEMS EBC-C413	Atom	Intel Atom E3845	1.92GHz	8GB	5.15.0

Table 7.2: Hardware platforms evaluated. *While the Raspberry Pi models tested support higher CPU clock speeds, we use the lower frequencies recommended in [26] and our prior work in [148, 152] to prevent throttling and instability.

7.7.1 Stage 1: Reconstruction

Reconstruction processes events independently, with its workload linear in n_r after a constant-time initialization. To characterize response time, we fit a linear function for each of our hardware platforms. We profile the reconstruction stage for values of n_r from 3000 to 27 000 in steps of 3000. For each value, we collect 20 response times from each of our 10 simulated GRBs. To better capture the worst case, we collect 200 times from each GRB for n_r=30 000. We fit a linear function over the maximum times for each n_r, then offset by the greatest positive residual to guarantee the function upper-bounds all 3200 observed response times. Measured worst-case times and characteristic functions are illustrated in Figure 7.8. Note that on the Atom, the extra samples for n_r=30 000 produced a slight outlier in measured WCET, indicating that the platform's timing is slightly less stable than the RPi3 or RPi4. Nonetheless, with our offsetting technique we were still able to meet all deadlines considered in Section 7.9.

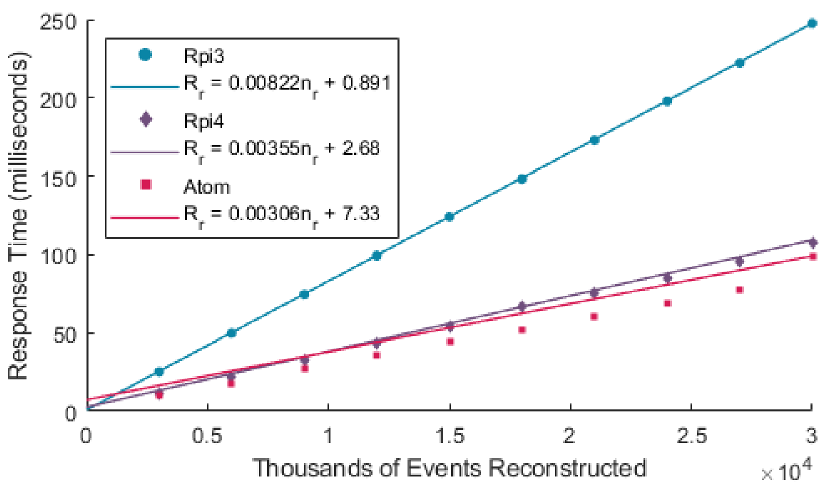


Figure 7.8: Reconstruction stage worst-case response times.

7.7.2 Stage 2: Approximation

Approximation initializes with results from reconstruction then samples annuli at random, with which it computes joint log-likelihoods for each of its candidate source directions. Aggregation and sampling are performed by a single subtask that precedes an independent subtask for each candidate source direction (100 for **FibSpiral** and 14 400 for **ApproxCircles**). A final subtask aggregates the results to find an average source vector, weighted by the joint log-likelihoods. Execution times for each subtask scale linearly with the number of sampled annuli n_s. We characterize the worst-case response time separately for each technique α.

We profile both approximation techniques for $n_s \in \{200, 400, 600, 800, 1000\}$. For each value, we collect 20 response times from each of 10 simulated GRBs. Similar to reconstruction profiling, we fit a linear function over the maximum times for each n_s, then shift vertically to guarantee an upper bound over all observed response times. The measured worst-case times and corresponding functions are illustrated in Figures 7.9 and 7.10. Notice that the vertical axis scale for **ApproxCircles** is 2 orders of magnitude greater than for **FibSpiral**, commensurate with the number of log-likelihood computations required by each technique.

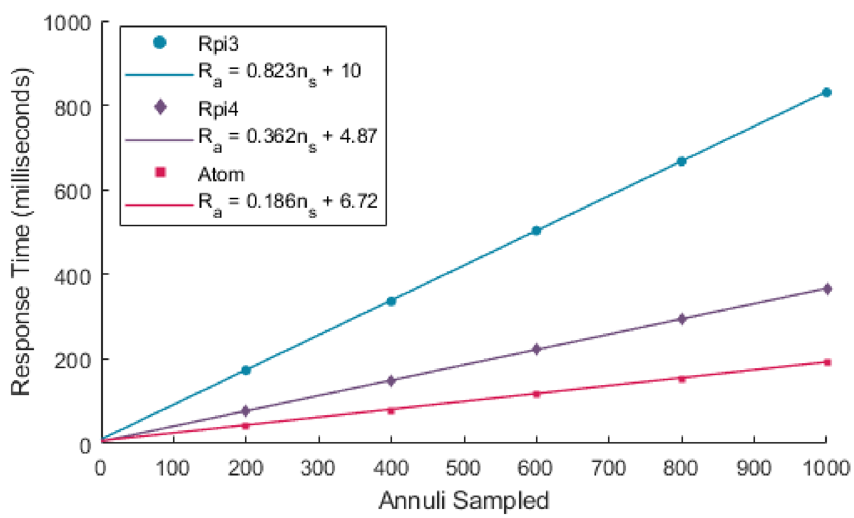


Figure 7.9: Approximation stage worst-case response times for **ApproxCircles**

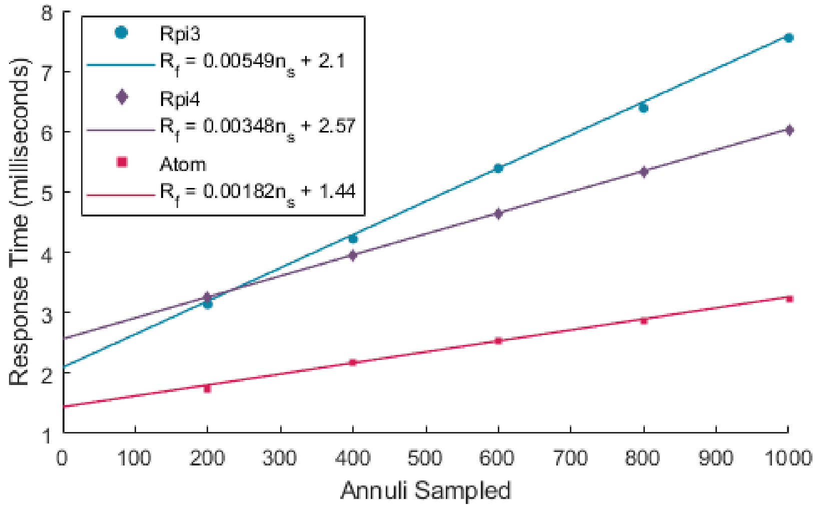


Figure 7.10: Approximation stage worst-case response times for **FibSpiral**

7.7.3 Stage 3: Refinement

Finally, **Refinement** is iterative; each iteration has a sequential initialization subtask followed by a parallel subtask to process and filter each annulus. A final sequential subtask in each iteration constructs and solves a constant-time quadratic eigenvalue problem, for which forming the matrix has cost quadratic in the number of reconstructed annuli [30]. Each iteration, then, has a worst-case response time quadratic in n_r; this is multiplied by x to produce the response time of the stage.

We fit this function on our three candidate devices, profiling each iteration of refinement from the same set of runs measured for reconstruction. Results are illustrated in Figure 7.11.

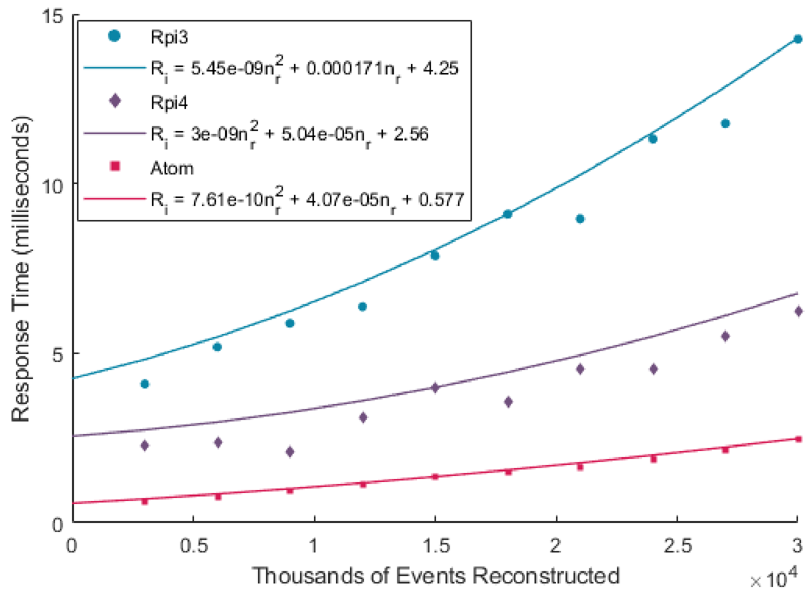


Figure 7.11: Refinement stage worst-case response times.

7.8 Implementation

In this section, we discuss our implementation of workload adaptation for APT's GRB localization task.

7.8.1 Offline Characterization of a Pareto-Optimal Surface

We quantified response times for 2657 input parameter states per the functions identified for each stage in Section 7.7. After generating a Pareto-optimal set of candidates according to Step ④ in Section 7.4.1, only 81 states remained for the RPi3, 84 for the RPi4, and 83 for the Atom.

As part of the characterization of Section 7.6, we quantified localization error for all values of α and x for each tested value of n_r and n_s, removing the option of interpolation over these discrete states. For each candidate state ξ, we reduced the hyperplanes connecting adjacent values of n_r and n_s to log-linear functions of error in each of these parameters. As the maximum value of n_r is a dynamic constraint, we also constructed log-linear functions from the points for which $n_r = 30\,000$ (the largest value tested) by extrapolation from the state with the next smaller value of n_r having a higher measured error. Each candidate state is saved in a data structure with its log-linear function parameters, and these are stored in a lookup table, implemented as a sorted array (`std::array`) to use online.

7.8.2 Online Adaptation

Determining Parameter Values

When a job arrives, corresponding to detection of a GRB, the localization task first computes the worst-case response time for the number of reconstructable events. If this exceeds the deadline, parameter values are selected by searching the lookup table according to Step ⑥ in Section 7.4.2.

Adapting to Overload

The number of events reconstructed affects the response time of the downstream refinement stage, so we do not treat reconstruction as an anytime workload. Instead, once parameter values are selected, global variables are set prior to computation to restrict the number of events reconstructed, the number of annuli sampled for refinement, and the number of refinement iterations to perform. The software implements **FibSpiral** and **ApproxCircles** as C++ subclasses of a common **Approximation** class, allowing dynamic object construction according to the chosen value of α.

7.8.3 Reclaiming Slack

When parameter values for an execution mode guarantee completion before the deadline in the worst case, pessimism in WCET estimates may result in overcompression. Nonetheless, some tasks provide opportunities for slack reclamation after computation completes.

The iterative source refinement stage of APT's GRB localization task is an anytime workload, so slack could be reclaimed naïvely by allowing it to continue iterating until the deadline.

However, it might still complete early if refinement converges, and this does not consider that earlier stages of the pipeline may also have been compressed. Instead, we implement a version of slack reclamation that determines, given the remaining slack time (less its own overhead), how many additional events can be reconstructed with another refinement iteration run over the resulting larger set of annuli.

For efficiency, rather than allowing OpenMP to split events among threads prior to reconstruction, an idle thread retrieves an event using an atomic fetch-and-increment of an index that tracks the next available event. Once this index reaches n_r, reconstruction halts, but is resumed when slack reclamation increases the limit. Reclamation continues in a loop until there is insufficient slack time remaining, as illustrated in Figure 7.12. At this point, additional iterations of refinement can still be run until the deadline (or until convergence).

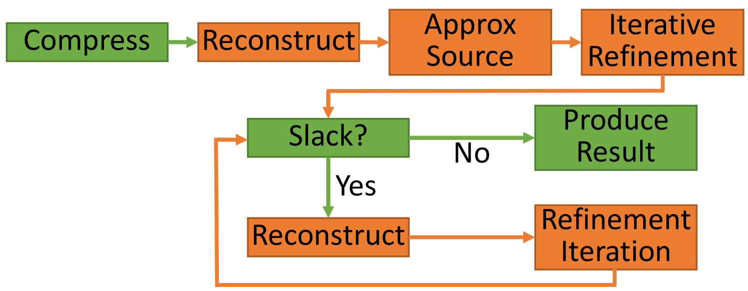


Figure 7.12: Localization pipeline with compression and slack reclamation.

7.9 Evaluation

In this section, we evaluate our proposed approach to parameterized workload compression for computationally-elastic, highly-parallel fork-join tasks. We begin by measuring the overheads associated with online adaptation to appropriately account for them, including searching the lookup table for a set of parameters, as well as the optional interpolation/extrapolation between states and slack reclamation. We next compare the accuracy of GRB localization when running with these optional enhancements against our synthetic bursts to decide whether they are actually expected to improve results. Finally, we evaluate our approach in the context of historical catalogued GRBs to demonstrate that our approach should generalize from the data used for initial characterization to the real-world, enabling localization of bright GRBs even under tightly-constrained deadlines.

7.9.1 Overheads

We begin with profiling the overhead of searching online for a Pareto-optimal set of compressed input parameters (Step ⑥ of Section 7.4.2) and of computing the inputs to slack reclamation (described in Section 7.8.3). We run the localization task against the synthetic GRBs described in Section 7.5, enabling both workload compression and slack reclamation. For each GRB, we test numbers of input gamma-ray photons over a geometric progression of 9 values from 30 to 10^6. Geometric progressions of 9 deadline values are selected separately for each hardware platform to guarantee that the shortest deadline would be between the response times of the first two candidate states, and that the longest deadline would be greater than the response time of the last candidate state.

Figure 7.13 illustrates the overheads of the 810 profiled runs of online compression for each tested hardware platform, with the horizontal bars enclosing the distribution. The overhead remained under 220 μs on the RPi3, under 180 μs on the RPi4, and under 60 μs on the Atom, ***demonstrating the efficiency of our online compression technique***. We adjust the response time functions for each of our platforms accordingly.

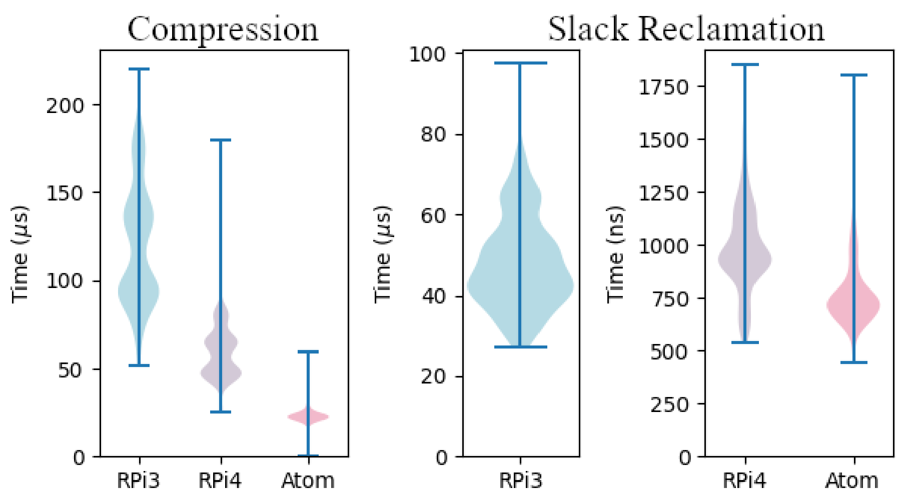


Figure 7.13: Measured overhead times.

Overheads associated with reclaiming slack are captured by profiling the elapsed time of the first successful attempt to reclaim slack for each run. Those runs for which slack could not be reclaimed are ignored, as the overhead may be lower in these cases. This produced 538 samples for the RPi3, 555 for the RPi4, and 573 for the Atom; these are also illustrated in Figure 7.13. Despite the equivalent program logic, the RPi3 demonstrated significantly higher overhead (notice the difference in vertical-axis units), reaching 97.5 μs. In contrast, the slack reclamation overhead remained under 1.9 μs on both the RPi4 and Atom.

7.9.2 Evaluation on Synthetic GRBs

To characterize the expected performance of our approach to elastic workload compression when applied to GRB localization on APT, we next evaluate three different implementations. The first, **Pareto**, finds the best state from the Pareto-optimal set of candidates with a response time that does not exceed the deadline and for which $n_r \leq n_e$ according to Step ⑥ in Section 7.4.2. The second, **IntExt**, additionally interpolates or extrapolates from that state. The third, **Reclaim**, performs compression equivalently to **IntExt** while also attempting to reclaim available slack time after the task completes according to the procedure outlined in Section 7.8.3.

We run each version to localize our 10 synthetic GRBs, using subsets of the generated gamma rays with sizes 10^N for N from 2 to 6. For each of the resulting 50 subsets, we evaluate the pipeline with a sufficiently large deadline to guarantee an uncompressed state, then imposed deadlines of 10, 33, 100, 330, and 1000 ms,[26] for a total of 300 sets of inputs to the pipeline. For each set of inputs, we run **Pareto** and **IntExt** once and ran **Reclaim** 5 times to account for variations in remaining slack time. We observed that over 1750 deadline-constrained runs on each of our three hardware platforms, no instance of the task missed its deadline.

To predict which approach is most likely to perform best on real-world datasets, we compare each approach pairwise with the other two. In this case, we define better utility as reducing localization error by at least 10%, because even a small change in input can result in significantly different results.[27] Results for all 1500 runs of **Reclaim** and 300 runs of **IntExt** and **Pareto** are illustrated in Figure 7.14.

[26] The shortest-duration burst captured by GBM was around 10 ms [71, 162, 23, 163].

[27] As reflected by the wide distributions shown in Figure 7.5.

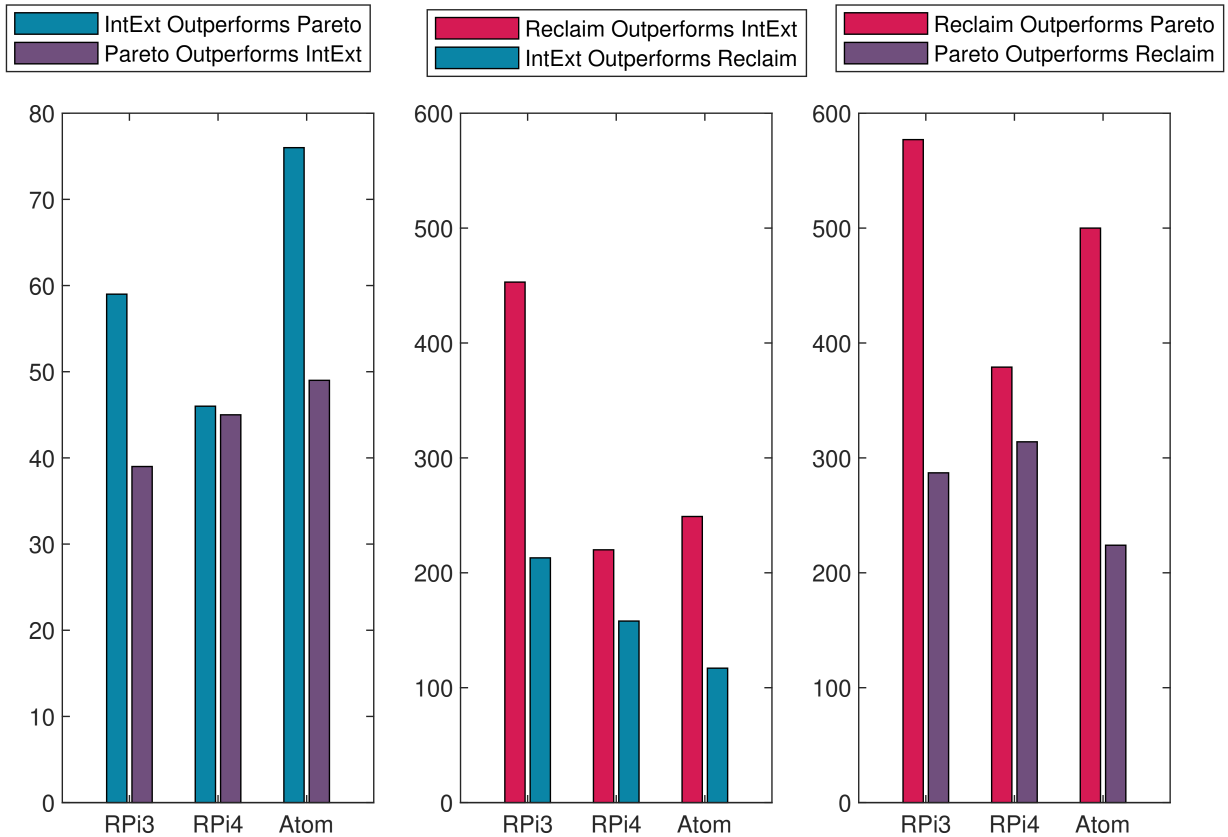


Figure 7.14: Pairwise comparison of approach versions for synthetic GRBs.

We note that a pairwise comparison provides more detail than enumerating the times each approach is the best of the three, which would not capture situations where two methods dominate the third, but not each other. We also observe that slack reclamation occasionally degrades results, as the additional input events selected might be incorrectly reconstructed or reflect noisy measurements, decreasing the fidelity of the final result. Nonetheless, the results suggest that ***additional interpolation or extrapolation from an initial candidate state, and reclaiming slack at the end of execution, are expected to improve outcomes*** most of the time.

7.9.3 Evaluation on Short GRBs Observed by Fermi GBM

To evaluate how well our approach extends from synthetic data to real-world workloads, we simulate four additional GRBs sourced from the Fermi GBM catalogs. We use the data in [113], which fits spectral-energy distributions to GRBs observed by the GBM. We searched for short GRBs (duration <1s) fit to a Band function; four matched these criteria. The corresponding simulation parameters are listed in Table 7.3. We randomly generate source directions by sampling the polar angle θ uniformly from 0–60° and the azimuth ϕ from 0–360°.

GRB	Δt	α	E_{peak}	β	Fluence	# Gamma Rays	θ	ϕ
80905499	0.704	0.66	284.6	-2.15	0.918	299 288	33.244	120.90
81209981	0.320	-0.67	1057.0	-2.25	2.452	818 489	40.576	43.60
90227772	0.704	0.48	2013.0	-3.15	20.272	1 772 628	16.766	37.64
90429753	0.832	-0.28	178.3	-1.65	2.643	803 322	31.572	214.17

Table 7.3: Simulated short GRBs with parameters matching corresponding catalog entries in [113]. Δt denotes the duration in seconds. E_{peak} is the peak of the energy spectrum in units of keV. Fluence is in MeV/cm^2.

Worst-case response times on each platform to perform uncompressed localization of each GRB are listed in Table 7.4.

Device	80905499	81209981	90227772	90429753
RPi3	1087	1379	5813	1177
RPi4	490	618	2808	529
Atom	265	346	1268	291

Table 7.4: Worst-case response times (ms) for uncompressed localization.

We run each implementation of our pipeline (**Pareto**, **IntExt**, and **Reclaim**) for each new GRB. We use a sufficiently large deadline to guarantee an uncompressed state, then impose a deadline equal to the burst's duration, and finally iterate over the same deadlines evaluated for our synthetic GRBs (10, 33, 100, 330, and 1000 ms). For each deadline, we run each version of the pipeline 20 times over each GRB, characterizing the 68% containment of error in source direction for each set of results. None of the 1440 deadline-constrained runs on each of our three hardware platforms missed its deadline.

Similarly to the analysis for the synthetic GRBs, we compare **Pareto**, **IntExt**, and **Reclaim** pairwise with the other two, enumerating how often each outperformed the others. Results are illustrated in Figure 7.15, with counts being out of 28 results. The results validate our predictions from the initial analysis of synthetic GRBs: ***interpolation and extrapolation from an initial Pareto-optimal state, then reclaiming slack time at the end of the pipeline, both typically improve localization accuracy.***

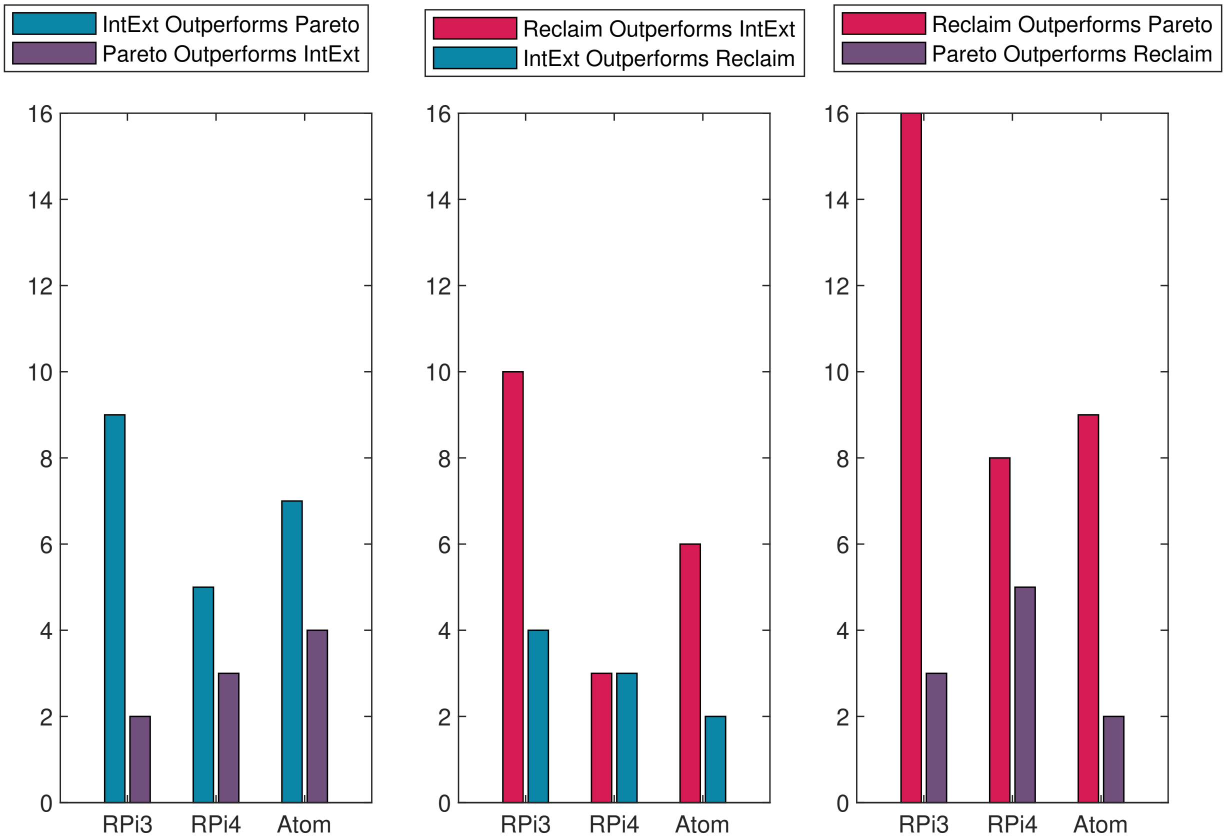


Figure 7.15: Pairwise comparison of approach versions for cataloged GRBs.

In Figure 7.16, we provide a plot for each simulated GRB, on each hardware platform, of the 68% containment of source direction error in degrees over the 20 runs of **Reclaim** for each imposed deadline.

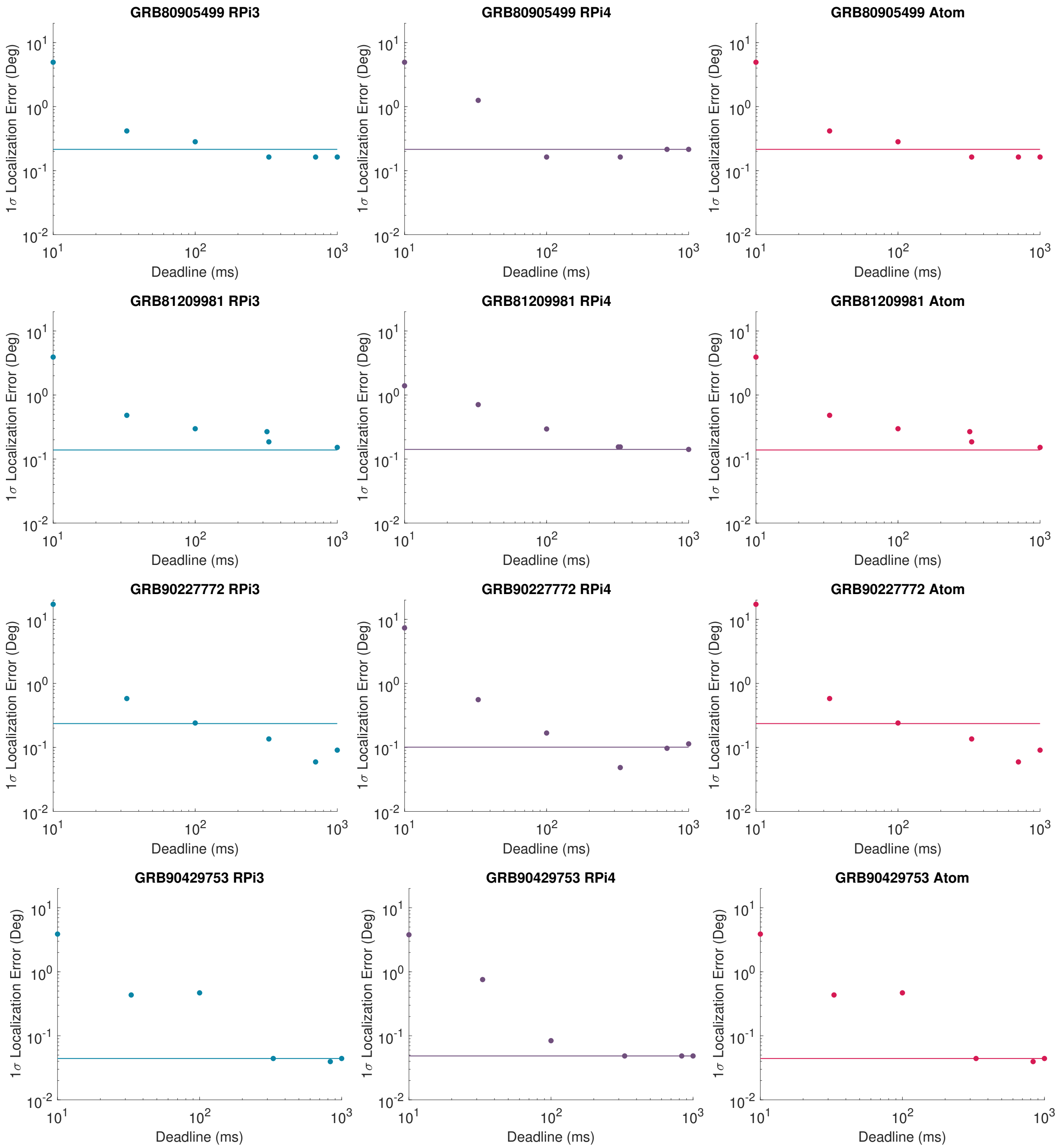


Figure 7.16: 68% containment of error in source direction using **Reclaim**. Horizontal lines indicate 68% containment for uncompressed execution.

We observe that, with compression, APT's GRB localization task is often able to produce results close in accuracy to those of its uncompressed state for deadlines of around 100 ms, even with worst-case uncompressed response times approaching 6 seconds on the platforms tested. While we allow for further compression to guarantee schedulability in response to dynamic workloads and deadlines, our approach also allows us to characterize a minimum acceptable deadline for each hardware platform. ***For deadlines as short as 33 ms, our techniques are often successful in providing sub-degree localization accuracy, sufficient for follow-up observations by many optical telescopes***. However, as the imposed deadline increases, localization error typically decreases, yielding greater utility. Nonetheless, this is not always the case: because of the high variance in localization accuracy, and because larger values of n_r might cause noisy or incorrectly reconstructed events to be selected, occasionally longer deadlines correspond to lower accuracy.

7.10 Conclusions

This dissertation has presented several new models under the framework of elastic scheduling to both extend the framework to new scheduling models. It has also considered the connection between how/where elastic constants are defined and the implications on system performance and result utility. In particular, the previous chapter proposed to apply elastic constants to each *subtask* of a parallel task to capture the individual impacts of reducing each of their workloads.

In this penultimate chapter of the dissertation, we have argued that our subtask-level elastic scheduling model in Chapter 6 is nonetheless insufficient. First, the quadratic cost function that Chantem et al. [44, 45] proved to be implied by Buttazzo's semantics of proportional compression [39, 40] may not reflect the actual cost associated with compression — the impact of degrading individual task or subtask workloads may be nonlinear or have higher-order effects in combination with other tasks or subtasks. Second, it treats compression of each system component independently. In Chapter 5, we demonstrated one example of interdependence — task periods might be constrained to harmonic values. Nonetheless, richer dependencies may arise; e.g., in a parallel task, decreasing the workload of one subtask might also decrease the workload of subsequent subtasks.

This chapter makes progress toward a richer model of elastic scheduling as the problem of parameterized workload compression to maximize utility within application-specific *and* schedulability constraints. Its contribution focuses on a narrower space of such problems, proposing a technique for compressing the workloads of highly-parallel fork-join tasks executing on a fixed number of processors to remain schedulable in the face of dynamic workloads and deadlines. The proposed technique identifies multiple discrete numeric, continuous numeric, and categorical parameters over which subtask workloads can be compressed, and through offline characterization of their effects on result utility and response times, a Pareto-optimal surface is generated to enable efficient online compression that guarantees schedulability while minimizing the resulting loss. It also identified methods to reduce pessimism by reclaiming available slack if execution completes early.

Inspired by the Astro2020 decadal survey [117], we applied this approach in the context of real-time GRB localization aboard the planned APT satellite mission. To enable rapid multi-messenger follow-up observations of transient astrophysical events, we demonstrated that ***APT can provide sub-degree estimates of source direction even for*** $\sim$33 ***ms deadlines imposed by bright, transient bursts.***

As a proof-of-concept, the results of this chapter demonstrate the need for a richer model. By using highly-parallel fork-join tasks on dedicated processor cores that are allocated a priori, characterizing response times is straightforward. Applying these techniques to more general DAG models, and considering the problem of processor allocation jointly among multiple parameterized tasks that may also have elastic periods, is a future research direction that naturally arises from the work of this chapter.

We expect this line of inquiry to continue to be shaped by the requirements of APT and other space-based time-domain and multi-messenger astrophysics missions. For example, the real-time properties of GRB localization might be better expressed with time utility functions, rather than a hard deadline: narrow observation windows may impose a tradeoff favoring earlier, but potentially less accurate, alerts [145]. Additionally, the GRB localization pipeline may run on shared hardware with mission-critical instrument control tasks (e.g., that regulate power or cool the instrument). Alternative analytical frameworks, such as semi-federated [78] or reservation-based federated scheduling [158] might allow these techniques to be extended to general parallel DAG tasks that share cores with low-utilization workloads. ***This work serves as a prerequisite toward a utility-driven elastic scheduling model over multiple tasks that share a limited set of resources.***

Chapter 8

Related Work, Conclusions, and Future Directions

In this chapter, we summarize the key conclusions of this dissertation, contextualize those within the body of existing literature, and motivate a broader vision for future research.

8.1 Scheduling Models

Among the primary contributions of this dissertation are new extensions of the elastic scheduling framework to existing scheduling models. To contextualize these contributions, we use this section to summarize the scheduling models already addressed by prior work on elastic scheduling and to show the scope of our extensions. Table 8.1 provides a visual overview while illustrating remaining gaps that we intend to fill in future work.

	Deadline									Multiprocessor					Elastic In				
Work	**Imp.**	**Con.**	**FP**	**EDF**	**Lock**	**Har.**	**Cmp.**	**Dis.**	**MC**	**Fluid**	**gEDF**	**PriD**	**Part.**	$\Vert$	T_i	C_i	$c_{i,j}$	D_i	**v.**
[39, 40]	●		●	●															P
[39, 40]	○			○	○														P
Ch. 2	●		●	●											●	●			P
[118]	●									●	●	●	●		●				P
Ch. 3	●									●			●		●	●			P
[44, 45]		○		○											○			○	QP
[13]		●		●											●				P
Ch. 4		●	●												●				P
Ch. 5	○		○	○		○									○				QP
[120]	●													DAG	●				QP
[119]	●													DAG		●			QP
[121]	●							●						DAG	●	●			QP
Ch. 6	●									●			○	DAG			●		QP
Ch. 7	●	●						●						F-J		●	●		G
[142]	○								○						○				–
[66]	○								○					DAG	○	○			QP
[132]	●		●	●			●								●				P
[133]	●		●	●			●						●		●				P

Table 8.1: Overview of elastic scheduling models. ● indicates an optimal algorithm (to within a tunable parameter ϵ), whereas ○ indicates a heuristic with opportunity for improvement. Column **Har.** indicates harmonic periods. Column **Cmp.** indicates compositional scheduling. Column **Dis.** indicates discretely-elastic tasks. Column **MC** indicates a mixed criticality system. Column $\Vert$ indicates the parallel task model under consideration — either general DAG tasks or highly-parallel fork-join (FJ) execution. Column **v.** indicates whether proportional compression (P) or the quadratic program (QP) of Chantem et al. [44, 45] is used, or if a more general (G) notion of loss is considered.

8.1.1 Implicit-Deadline Tasks on a Uniprocessor

The Original Elastic Scheduling Model

The elastic scheduling model of Buttazzo et al. in [39, 40] was developed for implicit-deadline tasks scheduled on a preemptive uniprocessor. Because it compresses task utilizations so that the total utilization does not exceed a given bound, the original model was applied to scheduling algorithms like EDF and RM for which utilization-based schedulability analysis provides an exact (or close-to-exact) test — this is the model already discussed in Section 2.2.2 of this dissertation. Buttazzo's proportional compression algorithm in [37, Figure 9.29] has execution time complexity $\mathcal{O}(n^2)$, where n is the number of elastic tasks.

Blocking due to Shared Resource Access

In the same work, Buttazzo et al. extended elastic scheduling to also consider resource constraints. When task execution includes critical sections of mutually-exclusive (i.e., locked) shared resource access, unbounded priority inversion might occur under standard scheduling algorithms. A famous example of this happened on the control computer for the Mars Pathfinder rover, which began experiencing total system resets within a few days of its 1997 landing on the Martian surface [79]. Investigation into this issue revealed that it was caused by a specific execution pattern related to access of the information bus. Access was synchronized with mutual exclusion locks (mutexes). To read from or write to the bus, a task would need to obtain a mutex first; another task requesting access would then be blocked until it could obtain the mutex. A high-priority task was responsible for bus management, and was invoked frequently to move certain types of data in and out of the information bus. However, a low-priority task responsible for gathering meteorological data would also use the information bus to publish its data, and would occasionally block the bus management task. The priority inversion occurred when a medium-priority communication task was scheduled during the brief interval that the high-priority management task was waiting for the low-priority meteorological task. The communication task, having a higher priority than the meteorological task, would preempt and prevent the meteorological task from making progress. This, in turn, prevented the high-priority management task from running. In this classic example of *priority inversion*, a task with high priority is prevented from making progress by a task with lower priority due to the semantics of shared resource locking.

Several protocols exist to bound the duration by which priority inversion may occur, and in our own prior work [148, 152], we have demonstrated how to implement such protocols in the CAmkES component framework [85] for the seL4 microkernel [134, 25]. Though the details differ, these protocols typically work by preventing a task that holds a lock from being preempted, if that preemption might cause additional priority inversion.[28] The stack resource protocol (SRP) is one such protocol [9], and it bounds the amount of time B_i that a task τ_i can be *blocked* by a task with a lower priority. A sufficient schedulability condition for implicit-deadline tasks is stated as:

$$\sum_{\tau_i} U_i + \max_{\tau_i} \left(\frac{B_i}{T_i} \right) \leq 1 \tag{8.1}$$

Buttazzo et al. [39, 40] note that elastic scheduling in the presence of shared resources becomes more complex, both because of the period in the denominator of the blocking term of the above expression, and because as periods change, the order may not be preserved, and therefore the maximum blocking time B_i for a task τ_i may also change. They therefore propose an algorithm that *overestimates* the blocking term by using a value B_i^{wc} that denotes the worst-case blocking time for any period order.

Our Contribution

In Chapter 2, this dissertation presented a new algorithm (Algorithm 2) for elastic scheduling of implicit-deadline tasks that ***compresses in time quasilinear in the number of elastic tasks***. Moreover, ***it runs in linear time when adjusting online to dynamic changes in system state***. By evaluating a large number of randomly-generated synthetic task sets, we observed that ***our algorithm achieves significant speedup during admission of new tasks***.

[28]Priority inversion naturally occurs already as the higher priority task waits for the lower priority task to release the lock, and this is typically unavoidable to guarantee consistency in whatever the lock is already protecting.

We constructed our algorithm to use task *utilizations*, rather than *periods*, as the parameter being adjusted; compression can therefore be realized by either increasing task periods ($T_i = U_i/C_i$) or decreasing task *workloads* ($C_i = U_i{\cdot}T_i$). This provides a straightforward extension to ***computationally-elastic tasks***.

Future Work

Two obvious questions remain unanswered from our work on developing an improved algorithm for elastic scheduling of implicit-deadline tasks on a uniprocessor.

First, ***can we use our approach to improve the algorithm that considers blocking times?*** A straightforward application of our algorithm arises from the overestimation of blocking times: from the schedulability condition in Expression 8.1, we can set $U_D = 1 - \max_{\tau_i}\left(\frac{B_i^{wc}}{T_i}\right)$, then apply our quasi-linear algorithm. However, there may be a bounded-time iterative approach that allows the real blocking times B_i to be used, even if they change in response to changes in priority order as periods are extended. We will investigate this in future work.

Second, ***when and how can workloads be compressed?*** If a task's period is increased while a job of the task is active, it is straightforward to extend the deadline of that job. If, however, the task's workload is decreased, this poses a challenge if a job of that task is active. We have discussed this question to some extent in Chapters 6 and 7 of this dissertation: a task might represent an anytime workload, in which case, early termination is appropriate. However, if the semantics of the task's computation require that its workload be set prior to execution, decreasing the workload might not be possible while a job is active. One simple option is to keep executing active jobs in a non-degraded state, and only decrease workloads for new job releases. This implies that the transition to the compressed state may take as long as $\max_i\{T_i\}$, which might be too long of a delay, depending on the circumstances under which elastic scheduling is being invoked. Another option is to extend the period (and deadline) while keeping the workload constant for *active* jobs, then adjust the workloads of jobs released after the transition. Identifying and evaluating applications for which these policies are appropriate is left to future work.

8.1.2 Sequential Tasks on Multiple Processors

Prior Work on Elastic Scheduling

In [118], Orr and Baruah extended elastic scheduling to sequential, implicit-deadline tasks running on multiple processors. Under *fluid* scheduling, as we explain in Section 3.2.1, task execution is abstracted to an assignment f of a fraction of a processor at each instant in time. So long as each task's utilization U_i individually does not exceed 1, the elastic model is equivalent to Buttazzo's original model, but with the utilization bound U_D set equal to the number of available cores.

Under *partitioned EDF* scheduling, as explained in Section 3.2.2, tasks are partitioned among processor cores such that each task's execution remains on one core. If the total utilization among tasks sharing a single core does not exceed 1 for each core, then the system is schedulable. Similarly to the approach in [13], Orr and Baruah propose in [118] to search for the amount of compression λ that achieves schedulability. They use bin packing heuristics to check schedulability for each value of λ tested.

Orr and Baruah also propose an elastic approach for *global EDF* (gEDF) scheduling [118]. Under global EDF, if there are m processors and the number of active jobs is greater than m, those m jobs with the earliest absolute deadlines are selected for execution. Schedulability is guaranteed if the following condition holds:

$$\sum_{\tau_i} U_i \leq m - (m-1) \times \max_{\tau_i} \{U_i\} \tag{8.2}$$

They again search for a value of λ, finding the smallest value with tunable precision ϵ for which the expression holds true.

Algorithm *PriD* proposes an alternative to gEDF to avoid the undesirable Dhall effect [53, 54] in which tasks with larger utilizations (close to, but less than, 1) may cause pathological configurations in which the utilization bound of gEDF becomes arbitrary close to 1, even for a large number of processors. Under PriD, if a task system is not gEDF schedulable, tasks are assigned dedicated processors in descending order of utilization until the remaining tasks are schedulable on the remaining cores according to Expression 8.2. In [118], Orr and

Baruah also apply their method of searching for an amount of compression λ that satisfies PriD schedulability.

Our Contribution

In Chapter 3 of this dissertation, we extended our improved quasilinear-time algorithm (Algorithm 2) to fluid scheduling. We also discussed and evaluated improvements to elastic scheduling for partitioned EDF. By changing the selection and order with which partitioning heuristics were applied, and by performing a binary, rather than linear, search for λ, ***we substantially improved the execution time associated with elastic scheduling for partitioned EDF tasks without significant overcompression.*** We also demonstrated an application of our improved algorithm to partitioned EDF by using the utilization bound guaranteed by the best-fit and first-fit heuristics. Though pessimistic, ***this approach achieved even greater speedups***, which may be necessary in mixed criticality systems where fast decisions must be made during overruns of critical tasks.

Future Work

As future work, we intend to consider whether ***similar improvements can be achieved for the global EDF and PriD algorithms***. Changing from an iterative search over the space of possible λ to a binary search is straightforward and provides an obvious improvement in execution time complexity.

The iterative approach used by Algorithm 2 may be hard to adapt for gEDF because U_i values appear in the schedulability bound (the RHS of Expression 8.2). However, we suspect there is a straightforward approach. Consider the task τ_j for which $U_j^{\max} = \max_i\{U_i^{\max}\}$, i.e., for which its utilization is the greatest when uncompressed. Then the schedulability condition can be rewritten as:

$$\sum_{i \neq j} U_i + m \cdot U_j \leq m \tag{8.3}$$

Solving for values of U_i should be straightforward by modifying Algorithm 2 to treat U_j as $m \cdot U_j$ and adjusting its minimum utilization and elasticity accordingly. Then, once utilizations are assigned in this way, if τ_j no longer has the maximum utilization, the algorithm can be rerun according to the new maximum. Work to formalize this algorithm and to prove a bound on execution time — i.e., how many times the algorithm has to rerun due to the task with the maximum utilization changing — are ongoing.

Applying such an approach to PriD is even less straightforward, because for a given amount of compression, the algorithm iterates over tasks to determine whether they should be assigned a dedicated core or scheduled globally. As future work, we will consider alternative approaches to obtain an exact solution for λ.

8.1.3 Elastic Scheduling as a Quadratic Optimization Problem

Chantem et al. [44, 45] extended elastic scheduling by showing that utilizations selected by the elastic model also solve a quadratic optimization problem, constrained by the minimum $U_i^{\min}$ and maximum $U_i^{\max}$ utilizations of each task τ_i, as well as by the utilization bound U_D afforded by the given scheduling algorithm. As we discussed in Section 4.2.1 of this dissertation, by changing or adding constraints to represent different schedulability conditions, the optimization problem serves as a template for elastic scheduling that allows it to be extended to alternative scheduling models or to solve related problems.

Constrained-Deadline Tasks

For example, in the same work, Chantem et al. replaced the constraint on total utilization with a set of constraints to represent the processor demand analysis (PDA) [20] schedulability test for constrained-deadline sporadic task systems under preemptive EDF scheduling on a uniprocessor. Due to the intractability of solving such a problem, for which the number of constraints may be very large, they also present a heuristic approximation [44, 45].

In [13], Baruah showed that these approximations are highly conservative and may result in significant overcompression for certain task sets. Two alternative approaches were presented that search for the amount of compression λ (defined in Equation 3.1 in Section 3.2.2 of this dissertation) to apply with some tunable degree of precision ϵ.

The Deadline Selection Problem

Chantem et al. [44, 45] use their quadratic program template to solve a related problem of *deadline selection* for constrained-deadline tasks. Under this model, a task $\tau_i = (C_i, D_i, D_i^{\min}, D_i^{\max}, T_i, E_i)$ is characterized by a workload C_i and *constant* period T_i. It has a desired deadline $D_i^{\min}$, but in the event that the constrained-deadline task set is not EDF-schedulable, each task's deadline may be increased up to its maximum value $D_i^{\max}$.

Chantem et al. present a similar heuristic to solve the deadline selection problem as the one they proposed for their dual problem of selecting *periods* while holding *deadlines* constant. Though they did not frame the problem as such, we will refer to tasks under this model as *deadline-elastic*; this is represented by a column in Table 8.1.

Our Contributions

In Chapter 4 of this dissertation, we extended uniprocessor elastic scheduling to fixed-priority, constrained-deadline tasks. We presented two algorithms that, similarly to Baruah's approach in [13] search for the amount of compression λ (with precision ϵ) needed to achieve schedulability, using both an iterative and a binary search over the space of possible values. ***Both approximate algorithms are highly efficient*** because they both track which tasks have already passed response-time analysis (RTA) for smaller values of λ so that repeated tests need not occur. Even for systems of 100 tasks and small values of ϵ, Elastic-FP-BS enables compression in under 20 milliseconds on a Raspberry Pi 3 Model B+.

We also presented a mixed-integer quadratic programming representation of the problem. Though solving a single problem jointly over all tasks is inefficient when using SCIP, an open-source solver [2], we presented an approach to solving the problem for *individual* tasks that nonetheless finds the *optimal* value of λ for *all* tasks. ***This MIQP-based approach is fast enough to feasibly perform compression offline.***

Future Work

As future work, we look to Table 8.1 for gaps that can be filled. ***Extending elastic scheduling to constrained-deadline, computationally-elastic tasks*** presents a unique challenge. It should be straightforward to apply the existing technique of searching for a value of λ. However, because the execution time terms C_i are not in denominators, and are outside of the floor ($\lfloor \cdot \rfloor$) for the demand bound function for EDF (Equation 4.2) and the ceiling ($\lceil \cdot \rceil$) for the recurrence relation in response time analysis for fixed-priority scheduling (Expression 4.11), other optimizations might be possible.

We would also like to consider an ***extension to deadline-elastic tasks***. Using similar techniques, we may be able to improve upon the heuristic proposed by Chantem et al. in [44, 45]. Furthermore, we would like to consider the case where ***deadlines must remain some fixed multiple of the task period***, so as periods are extended, so too are deadlines. Consider the case of a robotic assembly line, where a conveyor belt runs at some speed to deliver parts. Parts arrive according to the period T_i; because the belt is continuously moving, a control action by a robot along the line might have to be taken within some fixed time D_i of a part passing its work area. If the system is overloaded, the conveyor belt might be slowed down; in this case, due to the slower motion, both the period and deadline would decrease by the same proportional amount.

Finally, we are investigating an ***extension of elastic scheduling to constrained-deadline tasks on multiple processors***. Extending the MIQP from Section 4.6 to include processor assignments under partitioned scheduling should be straightforward, but may not be feasible to solve efficiency in SCIP. We will consider alternative solvers, such as Gurobi [72], with which we obtained good results in Chapter 6. Other techniques, such as Baruah and Fischer's polynomial-time algorithm for partitioning sporadic constrained-deadline task systems in [18], will be considered.

8.1.4 Harmonic Periods

Our Contribution

Chapter 5 of this dissertation presents the first work to extend elastic scheduling to task systems for which periods are constrained to be harmonic. To offer a tractable solution for online adaptation when available utilization changes, it considers a restriction of the problem where period assignments must maintain the same total ordering. ***This enables a polynomial-time algorithm that runs on the order of a few microseconds*** for online compression.

Future Work

We have shown that the harmonic period problem is at least as hard as integer factorization and that the harmonic elastic problem is at least weakly NP-hard. However, open questions remain about the complexity of these problems. We have demonstrated a pseudo-polynomial algorithm for the harmonic period problem, but we do not yet know whether it is weakly NP-hard. Furthermore, we do not know whether the harmonic elastic problem is NP-hard in the strong sense, and we have yet to prove a result about the ordered harmonic elastic problem.

Moreover, we still want to find ***an algorithm to solve the harmonic elastic problem in general, without an a priori order imposed on task periods***. We would also like to consider a version of the problem where *multiple* harmonic chains may be formed, with an a priori order imposed on each one. Consider, for example, dataflow applications where multiple front-end sensing tasks must be aggregated by back-end analysis or perception tasks. It might be a requirement that each sensor's period is an integer multiple of some back-end task, but that harmonic relationships do *not* need to be imposed among the sensor tasks themselves. Being able to solve a richer set of problems where harmonic relationships are expressed in graph form, with edges indicating harmonic relationships, and *directed* edges indicating that one task's period must be an integer multiple of the other (and not vice versa) is a future goal.

We also intend to address the challenges discussed in Section 5.6.2 associated with ***adjusting task periods in the middle of a hyperperiod***. As we showed, even if total utilization is compressed in response to a reduced schedulable bound, harmonic constraints mean that some periods will decrease. We intend to evaluate our two proposed policies — dropping jobs or extending task periods for a single hyperperiod — in the context of the real-time FIMS [166] and ORB-SLAM3 [42] applications (or others) that we evaluated in that chapter.

This might be made more complicated by scenarios where ***task workloads depend on the period relationships among tasks.*** For example, in a dataflow application, a back-end analysis task's workload might depend on how many sensor frames it processes at each invocation. For a given PHI (see Definition 4 in Section 5.5), the relationships between task periods remain constant, and so workloads should remain constant *within* that PHI. We therefore suspect that modifying the lookup table over optimal PHIs for each utilization bound interval will be straightforward for this scenario, but that a solution for dealing with execution time changes during transitions between states will be difficult to obtain.

Furthermore, we intend to extend our algorithm for the ordered harmonic period problem ***to multiprocessor scheduling***. Fluid scheduling provides a straightforward abstraction, removing the constraint that the total utilization bound remains less than 1. Global EDF scheduling may also be easy to adapt to tasks with harmonic periods. Again, for a given PHI, since relationships between task periods remain constant, the task for which the utilization U_i is the maximum in the RHS of the gEDF schedulability condition in Expression 8.2 should remain the same for any utilization bound. Efforts are ongoing to modify our algorithm that constructs a lookup table over optimal PHIs for each utilization bound interval such that the utilization bound reflects Expression 8.2 instead.

8.1.5 Federated Scheduling of Parallel Tasks

The federated scheduling model of Li et al. [94] deals with systems of *parallel* implicit-deadline real-time tasks. As described in 6.2.2, tasks are composed of subtasks representing individual sections of sequential execution, though multiple subtasks may execute in parallel. Execution must respect a precedence relation defined over the set of subtasks, which gives rise to the parallel DAG task model. Under federated scheduling, each parallel task executes on dedicated cores assigned to guarantee schedulability according to its workload, span, and deadline.

Prior Elastic Models for Federated Scheduling

In [120], Orr et al. extended the elastic framework to the federated scheduling model. If the total processor cores allocated exceed the number available, each high-utilization parallel task has its utilization compressed until the demand is met.

As a first attempt toward a solution, Orr et al. suggest finding the minimum amount of compression λ for which the system becomes schedulable according to the new allocation of cores to each task. *However*, Orr et al. argue in [120] that this may result in wasted capacity due to the ceiling operator ($\lceil\cdot\rceil$) used by the processor assignment taken from [94] that we listed earlier in this dissertation as Equation 6.2. If for some task τ_i the expression $(C_i - L_i)/(D_i - L_i)$ due to an assigned utilization $U_i(\lambda)$ is not an integer, then the period T_i could be reduced without affecting schedulability.

To avoid under-utilization, in [120] Orr et al. instead propose to assign utilizations according Chantem et al.'s quadratic optimization problem [44, 45]. Constraints for schedulability are updated to guarantee that the total number of cores that must be allocated to each task does not exceed the number available. In [119], Orr et al. extended their approach to *computationally-elastic* tasks, allowing parallel workloads to be adjusted over a continuous range: a task with period T_i would have its workload assigned as $C_i = T_i \cdot U_i$, but the span L_i is held constant, and the model does not address the question of how and from which subtasks workloads are to be reduced.

Orr et al. also point out in [121] that many tasks have discrete modes of execution. Rather than compressing task utilizations over continuous ranges of workload or periods, they propose a model of elastic scheduling under which a mode is selected for each task so that schedulability is guaranteed, and so that the objective function from Chantem et al. [44, 45] that assigns a cost associated with reducing each task's utilization is still minimized. In [121], Orr et al. apply an adaptation of a pseudo-polynomial dynamic-programming algorithm for the multiple-choice knapsack problem to this model.

Our Contributions

In Chapter 6 of this dissertation, we presented a new model of subtask-level computational elasticity for federated scheduling of parallel tasks. Each individual subtask has a workload that may take a value from a continuous range. Our model ***assigns an elasticity to each subtask*** to capture the individual impact of each subtask's workload on result quality. This also allows it to ***model the reduction in task span as subtask workloads are decreased***, which means it does not have to compress each task's total workload as much to remain schedulable on a given number of cores compared to the original model of Orr et al. for workload compression in [119].

We propose to solve the workload assignment problem that arises under this model by constructing an MIQP and solving it using Gurobi [72]. We demonstrate that this is feasible for offline compression. Moreover, by solving a set of MIQPs offline, ***we can achieve pseudo-polynomial online compression*** by finding an optimal way to allocate available cores to each task with dynamic programming (DP). This DP-based approach also enables ***the set of low-utilization tasks on the system to be considered jointly***, which was a limitation of the prior work.

Future Work

To solve the quadratic optimization problem that arises from our model of subtask-level elastic scheduling, we proposed two techniques for constructing the problem as a mixed-integer quadratic program (MIQP). While we demonstrated that by solving a collection of MIQPs for each task *offline* enables pseudo-polynomial time *online* compression, we cannot make any guarantees about the computational cost of obtaining a solution to the MIQP. Our proposed model is therefore unsuitable for use in real-time systems where the MIQP *cannot* be solved offline. This motivates us to explore alternatives to that model in the future.

As ongoing work, ***we are considering a return to the semantics of elastic scheduling via proportional compression of task utilizations*** as originally intended by Buttazzo's model [39, 40]. Though Orr et al. note in [120] that proportional compression of parallel tasks scheduled in a federated fashion results in wasted capacity, we argue that elastic scheduling should balance both *fairness* (degrading all subtask workloads in equal weighted measure, i.e., proportionally to their elasticity) and *utility* (we should seek to extract the best results that we can obtain — no subtask workload should be degraded more than necessary to ensure schedulability). To strike an appropriate balance, we can at once preserve the original semantics of elastic scheduling as closely as possible while avoiding wasted capacity by:

1. Compressing each subtask's workload proportionally to its elasticity until the entire task system is schedulable on the given number of cores.

2. Then every task τ_i now allocated m_i cores has its subtask workloads *decompressed* (still proportionally to their elasticities) as much as possible while still remaining schedulable on those cores. This remains as fair as possible (every task is allocated a number of cores based on proportional compression) without wasted resources (each task is compressed exactly as much as needed to fit on its core allocation).

We have developed both a mixed-integer linear program (MILP) to represent this problem, as well as an iterative algorithm to solve it. Efforts to evaluate both, and to prove an execution time bound for the iterative algorithm, are ongoing.

When considering the joint compression of low-utilization tasks, we outlined extensions of our approaches to fluid and partitioned EDF elastic scheduling from Chapter 3. For partitioned EDF scheduling, we suggested using the best-fit and first-fit heuristics to solve the bin packing problem to provide a pseudo-polynomial bound on execution time. Under the assumption that a collection of MIQPs might need to be solved offline for parallel tasks anyway, it may be worth the extra computation to instead use an exact schedulability test for partitioned EDF. In the same spirit as the other approaches presented in Chapter 6, we plan to formulate this problem as an MILP and evaluate its efficiency.

Though we considered the joint elastic scheduling of both high- and low-utilization tasks, we did not *not* consider hybrid-utilization tasks. Hybrid-utilization tasks were identified by Orr et al. in [120]; these are high-utilization parallel tasks that require more than one processor core in an uncompressed state. However, under elastic scheduling, these tasks may have their utilizations compressed until they become schedulable on a single core. The prior work of Orr et al. deferred the problem of dealing with such transitions; we also intend to consider such tasks in future work.

8.1.6 Mixed Criticality Systems

Tasks of multiple distinct criticality levels (e.g., mission- versus safety-critical) may execute concurrently on a shared platform [161, 32]. Criticality often corresponds to a required level of WCET certification: the standards for mission-critical components may be less stringent than those for safety-critical components. The adaptive mixed criticality model [16] characterizes each task according to a tuple $(T_i, D_i, \{C_i^\xi\}, \xi_i)$, where T_i and D_i are the usual period and deadline parameters, ξ_i indicates the criticality level, and C_i^ξ represents the task's WCET under the certification requirements of the corresponding criticality level ξ. This allows critical tasks to be characterized according to the more common-case execution times of lower certification levels, as well as the stricter worst-case times of higher levels. The system itself operates at a criticality level $\hat{\xi}$ (typically initialized to the lowest level). If an instance of a critical task executes longer than $C_i^{\hat{\xi}}$, system criticality is raised, and all tasks having $\xi_i < \hat{\xi}$ are temporarily dropped to guarantee completion of the more critical processes.

Earliest Deadline First with Virtual Deadlines (EDF-VD) is an optimal non-clairvoyant algorithm[29] for scheduling mixed criticality systems on a uniprocessor [14, 15]. Tasks are assigned dynamic priorities according to the EDF scheduling scheme, but high criticality tasks are scheduled according to a "virtual deadline," which scales each task's period by the ratio of the low-criticality utilization of all high-criticality tasks to the utilization available for those tasks.

Prior Elastic Models for Mixed Criticality Systems

Several alternatives to job and task dropping have been proposed. For instance, Su and Zhu's elastic task model for uniprocessor mixed criticality [142] characterizes each low-criticality task with a maximum period that reflects its minimum service requirement; if the system switches to a high-criticality state, periods are expanded to these values (with opportunity for early releases given sufficient slack). This model, however, is limited as an instantiation of Buttazzo's original elastic scheduling model. First, low-criticality tasks τ_i have their periods T_i fully extended to their maximum values $T_i^{\max}$, rather than degrading utilizations

[29]Non-clairvoyant in this context implies that the scheduler does not know, a priori, whether any task of a higher criticality will overrun one of its low-criticality execution times. The overrun is identified at the instant it occurs.

proportionally to their elasticities E_i until schedulability is guaranteed. Second, slack is reclaimed in a greedy fashion, rather than shared among low-criticality tasks in a manner that takes into account notions of fairness implied by their elastic constants. In fact, elastic constants are missing entirely from this model.

The adaptive mixed criticality model has also been extended to federated scheduling of parallel tasks [95, 96]. Under this model, tasks are additionally characterized with set of spans $\{L_i^{\xi}\}$, each certified according to the criticality level ξ. If a task overruns its execution time $C_i^{\hat{\xi}}$, the system level is raised and lower criticality tasks are dropped until sufficient processors have been freed for the critical tasks. As an alternative to pessimistic task dropping, Gill et al. [66] suggest that elastic compression may allow graceful degradation of less critical tasks without dropping them entirely. At a system criticality level $\hat{\xi}$, their model determines the total number of processors required by those tasks for which $(\xi_i \geq \hat{\xi})$. It then proposes compressing, rather than dropping, the less critical tasks to execute on the remaining cores.

Future Work

Though this dissertation does not make any contributions toward mixed criticality elastic scheduling models, it does point toward improved frameworks that leverage elasticity to provide graceful degradation in mixed criticality systems. For example, by ***assigning elastic constants to low-criticality tasks***, our quasilinear elastic scheduling procedure (Algorithm 2) might be used to decide the extent to which each task's period should be extended when the system criticality level increases. We are actively investigating these ideas in collaboration with others.

For federated scheduling of parallel tasks, we propose to ***apply subtask-level elastic scheduling to computationally-elastic, low-criticality tasks*** in the model of Gill et al. [66]. It should be straightforward to use the techniques presented in Chapter 6 to compress the workloads of low-criticality elastic tasks to execute on the number of available cores remaining after allocation to the high-criticality tasks.

Moreover, if workloads $c_{i,j}$ and workload constraints $c_{i,j}^{\min}$ and $c_{i,j}^{\max}$ are already being assigned at the subtask level, then a model naturally arises where worst-case execution times $c_{i,j}^{\xi}$ for each criticality level are also assigned at the subtask level. If subtask progress can be tracked, then the system criticality level may be increased when an individual subtask overruns at the current level. In this case, the core re-allocation to increase resources available to the critical task might be less pessimistic if the scheduler is cognizant of which of its subtasks have already completed execution.

As we mentioned earlier, it remains a challenge to decide how to realize workload compression, especially if it must occur immediately when the criticality level increases. If a job of a low-criticality task is already executing, and it does not have anytime semantics, then reducing its computational budget might not guarantee any result, even a degraded one. Nonetheless, this scenario is no worse than otherwise dropping the job in the non-elastic version of adaptive mixed criticality.

8.1.7 Compositional Scheduling

Compositional real-time systems are those for which multiple schedulers are composed, often in hierarchical fashion, to provide different scheduling semantics for applications with unique timing requirements that must nevertheless execute concurrently on the same system [137]. Though our dissertation does not consider compositional systems, we aim provide a complete picture of the scheduling models to which elastic scheduling has been extended.

Prior Elastic Models for Compositional Systems

In [132], Salman et al. apply the elastic framework to uniprocessor compositional scheduling. They consider a system composed of multiple applications Γ_i, each providing its own RM or EDF scheduler. A system-level scheduler uses the periodic resource model of Shin and Lee [138] under which each application-level scheduler is given a reservation $\Gamma_i(\Theta_i, \Pi_i)$ of Θ_i time units on the CPU every Π_i time units. Schedulability is determined according to two conditions: *(i)* the total resource supply must be sufficient for the reservations given to all application-level schedulers, and *(ii)* application's tasks must be schedulable given the provided reservation, according to the condition of Shin and Lee [138]. In their work in [132], Salman et al. use a modified version of Buttazzo's algorithm in [39, 40] to determine, for a fixed resource supply, how the elastic application can adapt its frequencies to remain schedulable.

In [133], Salman et al. extend this idea to compositional scheduling on a multi-processor system. Instead of the periodic resource model, they use the Minimum Parallelism Supply model (MPS) from Leontyev and Anderson [89]. Under this model, an application-level scheduler is assigned a set of dedicated cores, as well as a reservation $\Gamma_i(\Theta_i, \Pi_i)$ on a shared processor. The proposed approach uses the technique of Orr et al. in [118] for partitioned EDF scheduling, iteratively increasing the amount of compression λ and heuristically partitioning tasks among processors. They modify the partitioning heuristic slightly to also assign tasks to the shared processor, and check schedulability on that processor equivalently to the uniprocessor case they considered in [132].

Future Work

Though we have not considered compositional scheduling directly in this dissertation, the applications we have evaluated hint at a compositional scheme. In Section 5.7, we evaluate applications of our harmonic elastic scheduling model to both the real-time FIMS [166] and ORB-SLAM3 [42] pipelines. In these evaluations, we consider the situation where the target application executes on a resource-constrained system concurrently with other applications that may limit the amount of processor utilization available to the target. A more principled representation of this idea would be to use compositional scheduling, with each application subsystem given its own reservation over the processor resources. In such systems, the elastic model could be applied at two levels: within the reservation granted to the application (as we demonstrated in our evaluation), and to allocate resources among application subsystems at the top level.

8.1.8 Other Scheduling Models to Consider

It would be infeasible to discuss all models that arise from combinations of columns in Table 8.1; we could discuss at length the ways in which elastic scheduling might be extended, e.g., to constrained-deadline tasks with harmonic periods, or to fixed-priority shared resource access protocols. We instead focus here on a few more natural extensions of the work in this dissertation, especially in the context of parallel task scheduling.

Chapter 7 of this dissertation presents a vision of elastic scheduling as a richer model of parameterized workload adaptation. It presents an approach for highly-parallel fork-join tasks, for which it is straightforward to express the exact response time as a function of subtask workloads (see Equation 7.3 in Section 7.4.1). However, for DAG tasks in general, finding the minimum possible response time on a given number of cores (or, equivalently, determining the minimum number of cores to meet a given deadline) is NP-complete [159]. As a result, the models of Orr et al. [120, 119, 121], as well as our model in Chapter 6, use the generally *sufficient*, but not *optimal*, core assignment of Equation 6.2.

However, ***there are pseudo-polynomial algorithms that build schedules based on the structure of the DAG*** [55], including an algorithm we proposed in prior work [150] that often achieves an optimal processor assignment. It would be straightforward to implement in the context of period-elastic parallel tasks: using the algorithm to construct a schedule for a given number of processors gives us a response time that defines the minimum period that can be assigned. This gives rise to a DP-based approach where each task has a period (and therefore, a utilization and corresponding objective function value) assigned for each possible core allocation; the problem then is to find an optimal joint allocation of cores to all tasks.

An alternative method is to ***embed the DAG scheduling problem into the MIQP itself.*** In the absence of elasticity, several techniques exist to construct MILP representations of DAG scheduling problems, with subtask precedence relations encoded as constraints. We will not attempt to provide exhaustive coverage of such techniques here; instead, we refer the reader to the introductory sections of [160].

Nevertheless, such models are still constructed according to the federated scheduling paradigm, under which ***high-utilization parallel tasks are assigned dedicated processor cores***. On embedded systems, such as the quad-core computational platforms considered for use onboard our proposed APT satellite, this may result in unnecessary resource waste, since other tasks (such as networking, instrument control, telemetry, etc.) will need to execute concurrently. Alternative analytical frameworks, such as ***semi-federated scheduling*** [78] or ***reservation-based federated scheduling*** [158] could allow low-utilization sequential tasks to share processors with parallel tasks. Moreover, an elastic model for reservation-based federated scheduling would support ***constrained-deadline parallel tasks***. We intend to investigate extensions of the elastic scheduling framework to these models as future work.

As another alternative to the federated scheduling model for parallel DAG tasks that we want to consider, Wasly and Pellizzoni's ***bundled scheduling*** [170] provides an abstraction where tasks are divided into sequences of segments called bundles, each requiring a different number of threads. Threads forming a bundle are scheduled together, with each thread assigned its own core. However, this abstraction allows the number of cores assigned to a parallel task to change as it executes, reducing the length of idle intervals and providing an improved resource allocation strategy. An elastic model for bundled scheduling may better reflect the semantics of many common parallel operations, such as matrix multiplication. For example, compressing a task's workload by reducing the size of a matrix would actually reduce the number of subtasks under a DAG model, but the existing traditional elastic models (besides our parameterized model in Chapter 7) for federated scheduling assume that the task's DAG structure remains the same under compression, even if its individual subtask workloads may change. Under a bundled scheduling model, this would simply reduce the execution length of existing bundles.

Another challenge that arises in parallel task scheduling is modeling the overheads associated with cache access, thread synchronization, and communication costs between cores. In [63, 64], Ferry et al. showed that even for the highly-parallel matrix-vector operations used in numerical models for real-time hybrid simulation (RTHS), execution times do not scale inversely with thread counts. Rather than needing a closed-form expression to capture these effects, a bundled elastic scheduling model could be constructed as a problem of selecting from multiple bundle configurations associated with each task phase. Bundle configurations could reflect both compressed workload configurations, as well as their execution times when scheduled using different numbers of threads. Constructing such a model, finding algorithms to solve it, then applying it to GRB localization and a new benchmark problem for multi-axial RTHS [49] are topics of future work.

8.2 Applications of Elastic Scheduling

As we have shown, elastic scheduling can be realized to provide a framework under which applications can adapt their execution in a principled way. In this section, we describe prior work in this direction, state conclusions drawn from our contributions, and discuss future research directions.

8.2.1 In the Prior Work

Control Systems

Buttazzo et al. state that elastic scheduling is readily applied to multimedia systems, in which "timing constraints can be more flexible and dynamic than control theory usually permit" [40]. However, prior literature exists (including later works of Buttazzo's) that considers leveraging elasticity in task periods and workloads for adaptive control systems.

In their papers on ***adaptive rate control*** [34, 35], Buttazzo and Abeni apply elastic scheduling to adapt periods in response to dynamic system load. They argue that in some systems, task execution times might only be estimated, not characterized with worst-case values. They propose a control loop to continuously update estimated workload values C_i based on a runtime monitoring mechanism. Task budget reservations are adjusted in response, and if these would cause the system to be overloaded, task periods can be adapted according to the original elastic algorithm in [39]. They suggest that these techniques can be applied to control applications for which periodic tasks may execute at different rates, depending on the operating conditions; for example, in avionics, the minimum safe altimeter sampling rate is a function of the aircraft's altitude.

Buttazzo et al. also propose to use elastic scheduling to ***manage the quality of control (QoC) in overloaded systems*** [36]. The major problem addressed by the paper arises from the fact that digital controllers that run at a given frequency have control laws that are tuned to that frequency. However, in systems where overload may occur — e.g., due to execution times increasing beyond their estimated values (as in the adaptive rate control papers) or when new tasks are activated due to changes in the environment — task rates may have to adapt to maintain schedulability. The paper primarily addresses the problem of selecting multiple controllers that can be switched out depending on the period assigned to the corresponding task. The goal is to select, for each control task, a set of controllers from which a selection minimizes control error over a range of acceptable periods. Too many controllers increases the memory footprint of the application, whereas too few controllers may deviate too much from the invocation frequencies to which they are tuned, increasing the associated control error. The problem of *period assignment* is addressed according to Buttazzo's elastic scheduling model, with elastic coefficients "set to reflect tasks' importance" [36]. This work does not explicitly consider ***codesign of the elastic scheduling constants with the control error functions under consideration***.

Tian and Gui attempt to bridge this gap in [155] by ***embedding both the QoC management and elastic workload adaptation into a constrained optimization problem*** à la Chantem et al. [44, 45]. They propose to assign elastic constants according to the inverse of the instantaneous weighted QoC of an individual control loop. This decision is somewhat heuristic, and aims to represent control cost as a function of task utilization so that the objective functions match. Nonetheless, it provides a more principled assignment of elastic constants than simply a qualitative notion of "importance."

Dynamic Voltage Scaling

In embedded systems for which energy consumption is a concern, power draw can be reduced by scaling down voltage, which decreases processor speed. In the model of Marinoni and Buttazzo [104], the execution time C_i on a speed s processor ($s \leq 1$) for a task τ_i is given by

$$C_i(s) = \frac{\phi_i C_i^{max}}{s} + (1 - \phi_i) C_i^{\max}$$

where $C_i^{\max}$ is its execution time on a unit-speed processor and ϕ_i is an application-dependent constant indicating the proportion of CPU-bound code.

Marinoni and Buttazzo consider the situation where ***a power management policy has set the processor speed*** to s. They demonstrate how to determine whether the system is now overloaded, and if so, they apply Buttazzo's elastic scheduling model [39, 40] to adjust task periods. They demonstrate how to calculate the utilization constraints $U_i^{\max}(s)$ and $U_i^{\min}(s)$ as functions of the processor speed, then they compress utilizations according to task elastic constants using the usual algorithm.

QoS Management in Distributed Systems

Pedreiras and Almeda point out in [124] that CPU utilization is not the only limited resource over which usage must be scheduled. In real-time distributed systems, which (even in 2003 when the paper was written) are becoming increasingly pervasive in avionics, automotive, robotics, computer vision, and multimedia applications, available ***network data rates*** must be shared among multiple message sources. The authors propose to apply the original elastic scheduling model of Buttazzo et al. [39, 40] to compress network utilization in real-time networking protocols. They apply this idea in a case study with a mobile robot.

8.2.2 Applications Considered in This Dissertation

Atmospheric Particle Characterization

In Section 5.7.1 of this dissertation, we applied our model of elastic scheduling for tasks with harmonic periods to the Fast Integrated Mobility Spectrometer (FIMS) [167, 169], a flown instrument that characterizes atmospheric aerosol particle size distributions. Real-time aerosol sampling may allow adaptive flight patterns for atmospheric survey missions. However, task workloads may vary with the density of the sampled particles, the vibrations of the enclosing chamber, and the changing reflectivity of the chamber walls [169]. Moreover, if deployed on SWaP-constrained hardware atop a lightweight UAV, the FIMS computational pipeline may need to share resources with other applications, such as path planning, localization, and aircraft control. The amount of processor utilization available to the FIMS application may therefore fluctuate as environmental conditions change. Toward providing an adaptive framework atop which FIMS can execute, we demonstrated that our elastic model enables FIMS to adjust its task periods to avoid missing deadlines when run concurrently with a dynamic, high-priority interference task.

Simultaneous Localization and Mapping

In Section 5.7.2 of this dissertation, we applied our model of harmonic elastic scheduling to a simultaneous localization and mapping (SLAM) system [88], where harmonic periods help to enforce frame alignment from multiple sensor sources. In particular, we considered ORB-SLAM3 [42], which is a visual-inertial SLAM system widely used in autonomous vehicle and robotics applications that supports stereo camera inputs. Again, ORB-SLAM3 may run concurrently with other application subsystems (e.g., in a future deployment, it might run alongside FIMS), so providing an adaptive framework under which it may change task rates if available utilization changes is an important goal. While prior studies (including one by our collaborator Ao Li [90]) on adapting SLAM systems already exist, these typically execute by selectively dropping data frames, and often require highly-tailored ML-based integration with the existing application. We demonstrate that our elastic framework provides even better results (i.e., less degradation in localization accuracy) than the approach in [90] by simply adjusting task periods in response to overload.

GRB Localization

In Chapter 7, we considered elastic workload adaptation for the task of real-time gamma-ray burst (GRB) localization onboard a future satellite telescope. We argued that a richer model of elastic scheduling, framed as an optimization problem over multiple parameterized degrees of freedom within the constraints of schedulability, is needed to fully capture the semantics of adapting the application's workloads. Our model captures the dimensions — continuously numeric, discretely numeric, and categorical — over which workloads can be compressed, which also captures the interdependencies among subtasks: changing the workload of one subtask (e.g., by reducing the input data) may also change the workload of a downstream subtask. Rather than attempting to characterize a closed-form expression for utility as a function of workloads, through extensive experimental evaluation offline, the proposed technique constructs a surface over which online search allows rapid adaptation in response to changing environmental conditions.

The Astro2020 decadal survey [117] released by the National Academies identifies "space-based time-domain and multi-messenger program" as the highest-priority sustaining activity in space, which will require coordinated real-time follow-up observations of transient astrophysical phenomena – e.g., gamma-ray bursts (GRBs) — using secondary observational modalities, such as visible-light observations with optical telescopes. Pursuant to this, we applied our adaptive framework to determine whether we could achieve accurate real-time localization aboard a simulated model of our proposed orbital Advanced Particle-astrophysics

Telescope (APT). Even with the dynamic workloads associated with the varying brightness and spectral parameters of GRBs, and even under very short (33 ms) deadlines imposed by the possible future requirements associated with prompt localization, we are able to achieve subdegree estimates of GRB source directions.

8.2.3 Future Directions for Control Applications

As future work, we intend to investigate formal relationships between control performance and elastic scheduling. The prior work in this regard remains limited. As we mentioned, Buttazzo et al. [36] try to optimize the availability of controllers corresponding to different invocation frequencies, which can be selected as task periods change under Buttazzo's elastic scheduling model. However, task utilizations are compressed per elastic constants assigned according to the application designer's notion of importance. Tian and Gui [155] argue that the control cost associated with increasing task periods should be encoded into the elastic constants, such that utilization compression optimizes control performance within the constraints of schedulability. Nonetheless, there is still fertile ground for future investigation: Tian and Gui pick a representation of control cost to match the quadratic optimization of Chantem et al. [44, 45]. We propose that the opposite approach should be taken, with an objective function chosen such that it represents, as closely as possible, classical expressions of control cost.

In our own application of elastic scheduling to ORB-SLAM3 in Section 5.7.2 of this dissertation, we assigned elastic constants according to the first-order impact on localization accuracy associated with reducing each task's sampling rate from its nominal frequency. Though this is a principled way to couple the semantics of elastic scheduling identified by Chantem et al. [44, 45] to one notion of control performance, it remains a strictly empirical method: the values were obtained from experimental results. Again, we propose that in the future, a deeper formal connection between control performance and sampling/processing rates in SLAM systems should be quantified, which will make elastic models even more robust to different environments and resource constraints.

8.2.4 Future Directions for Localization of Astrophysical Transients

Time-domain and multi-messenger astronomy requires the coordination of multiple geographically disperse instruments for both detection and direct follow-up observations. Work to model this as a distributed real-time system, with components that must work together to observe transient phenomena within short windows of opportunity, is ongoing.

Recent advances in fast-slewing robotic optical telescopes, such as the superfast TURBO telescopes under development [116], offer significant opportunities for more rapid and accurate follow-ups. To fully harness the potential of those new fast-slewing instruments, important new research questions at the intersection of real-time control and scheduling must be addressed. For example, orienting an instrument as accurately as necessary, but as quickly as possible to best observe an emerging and potentially short-lived phenomenon, requires rapid evolution and convergence of control objectives, even as newly sensed data demands iterative refinement of control parameters. Furthermore, the instrument must be protected from damage or misalignment that may result from, e.g., excessive accelerations during higher-performance maneuvers to orient the instrument.

Defining the tradeoffs between *accuracy* and *timeliness* of localization as they affect the objective to *maximize the expected observations that can be performed* while staying within the *safe regions defined by the necessary control actions* will require the coupling of elastic scheduling theory with control. It will also likely lead to models that integrate other concepts such as time-utility curves [156] (to capture how the usefulness of follow-up observations evolves over time, instead of applying strict deadline constraints) and scheduling with explorable uncertainty [56] (to capture the ability of follow-up instruments to search a broader region to which the localization algorithms bound the direction of the phenomenon).

8.3 Open Questions and Broader Vision

Cyber-physical systems are becoming increasingly pervasive in dynamic environments where they must be able to *adapt* to changes in resource availability or external conditions to maintain *temporal* correctness while still remaining functional. Elastic scheduling has emerged in the last two decades as a framework under which such adaptation is realized.

This dissertation has made several contributions to the state of the art in elastic scheduling models. It has presented new algorithms for parameter assignment with improved execution time complexity, enabling fast adaptation in response to online changes, such as during admission of new tasks or when computational resources change. It has extended elastic scheduling to new task models, enabling it to be used in constrained-deadline fixed-priority task systems and systems of tasks with harmonic rate constraints.

It has also considered what it means to be elastic, and how to assign elastic parameters in a principled way. It used the quadratic optimization problem that arises from Buttazzo's model of elastic scheduling to define elastic constants according to the first order impact on result error associated with decreasing task rates from their nominal values. Using this technique, it implemented elastic versions of aerosol sampling and autonomous vehicle localization and mapping applications. It also presented a new model of subtask-level computational elasticity, suggesting that workload compression should be considered for individual subtasks within a parallel task. It discussed the implications of such a model with respect to the DAG structure that arises from precedence constraints among the subtasks, and the subsequent schedulability analysis.

These extensions have all stayed within the semantics of the original elastic scheduling model that proposes to proportionally compress task utilizations, or the dual quadratic optimization problem that minimizes their weighted squared deviations. However, this dissertation also argues for a more general model of elastic scheduling as ***the reduction of task utilization along multiple parameterized degrees of freedom so as to remain within the constraints of schedulability while minimizing loss in result utility.*** Under such a model, task utilizations are still compressed, but compression is no longer required to be proportional to a single constant. This allows models to capture more complex interdependencies and nonlinearities in the relationships between task utilizations and outcomes. Furthermore, it allows a better relationship of the relationships between subtasks: reducing the workload of one subtask may naturally change the workload of another. We applied this approach to gamma-ray burst localization aboard a future orbital telescope, and demonstrated that it enables localization to adapt in the face of unknown burst parameters and short dynamic deadlines without significant loss in accuracy.

As future work, we intend to continue to apply the traditional elastic scheduling models to new scheduling frameworks that are already commonly used in real-time systems. We also intend to expand on our ideas of elastic scheduling as a broader framework, and consider more formal ways to represent and characterize tasks within that framework. We will also apply the framework to systems of *multiple* tasks — at present, our work is restricted to parameterized workload compression for a single highly-parallel fork-join task.

We also intend to address open questions related to transitions between compressed states. Under the original elastic model, task periods are extended in response to overload; active jobs simply have their deadlines extended in response. However, transitions are not so straightforward in other models. For example, when periods are restricted to harmonic values, decreases in available utilization might result in one or more task periods also *decreasing* to maintain integer ratios among period values. Furthermore, the workloads of computationally-elastic tasks may decrease in response to system overload; the execution time budget allocated to currently executing jobs therefore may be insufficient if a computational mode has to be selected at the beginning of the program. While we have suggested policies to deal with such scenarios for different system and program semantics, implementing and evaluating those policies remain necessary for better adoption in real systems.

References

[1] J. J. Abbott, Z. Nagy, F. Beyeler, and B. J. Nelson. Robotics in the Small, Part I: Microbotics. *IEEE Robotics & Automation Magazine*, 14(2):92–103, 2007.

[2] T. Achterberg. SCIP: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, Jul 2009.

[3] S. Agostinelli, J. Allison, K. Amako, et al. Geant4 — a simulation toolkit. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 506(3):250–303, 2003.

[4] S. Aldegheri, N. Bombieri, D. D. Bloisi, and A. Farinelli. Data Flow ORB-SLAM for Real-time Performance on Embedded GPU Boards. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5370–5375, 2019.

[5] B. Andersson and D. de Niz. Analyzing Global-EDF for Multiprocessor Scheduling of Parallel Tasks. In R. Baldoni, P. Flocchini, and R. Binoy, editors, *Principles of Distributed Systems*, pages 16–30, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[6] N. Audsley, A. Burns, M. Richardson, and A. Wellings. Hard real-time scheduling: The deadline-monotonic approach. *IFAC Proceedings Volumes*, 24(2):127–132, 1991. IFAC/IFIP Workshop on Real Time Programming, Atlanta, GA, USA, 15-17 May 1991.

[7] Y. Bai, L. Li, Z. Wang, X. Wang, and J. Wang. Performance optimization of autonomous driving control under end-to-end deadlines. *Real-Time Systems*, 58(4):509–547, Dec 2022.

[8] Y. Bai, Z. Wang, X. Wang, and J. Wang. AutoE2E: End-to-End Real-time Middleware for Autonomous Driving Control. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 1101–1111, 2020.

[9] T. P. Baker. Stack-based scheduling for realtime processes. *Real-Time Syst.*, 3(1):67–99, 4 1991.

[10] D. Band et al. BATSE Observations of Gamma-Ray Burst Spectra. I. Spectral Diversity. *Astrophys. J.*, 413:281, Aug. 1993.

[11] I. Bartos and M. Kowalski. *Multimessenger Astronomy*. 2399-2891. IOP Publishing, 2017.

[12] S. Baruah. Partitioned EDF scheduling: a closer look. *Real-Time Systems*, 49(6):715–729, Nov 2013.

[13] S. Baruah. Improved uniprocessor scheduling of systems of sporadic constrained-deadline elastic tasks. In *Proceedings of the 31st International Conference on Real-Time Networks and Systems (RTNS 2023)*, New York, NY, USA, 2023. Association for Computing Machinery.

[14] S. Baruah, V. Bonifaci, G. DAngelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. The Preemptive Uniprocessor Scheduling of Mixed-Criticality Implicit-Deadline Sporadic Task Systems. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 145–154, 2012.

[15] S. Baruah, V. Bonifaci, G. D'angelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie. Preemptive uniprocessor scheduling of mixed-criticality sporadic task systems. *J. ACM*, 62(2), 5 2015.

[16] S. Baruah, A. Burns, and R. Davis. Response-Time Analysis for Mixed Criticality Systems. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 34–43, 2011.

[17] S. Baruah and P. Ekberg. An ILP representation of response time analysis. Short note available from https://research.engineering.wustl.edu/~baruah/Submitted/2021-ILP-RTA.pdf, 2021.

[18] S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems. *IEEE Transactions on Computers*, 55(7):918–923, 2006.

[19] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, Jun 1996.

[20] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Syst.*, 2(4):301–324, oct 1990.

[21] K. Bechtol, S. Funk, A. Okumura, L. Ruckman, A. Simons, H. Tajima, J. Vandenbroucke, and G. Varner. TARGET: A multi-channel digitizer chip for very-high-energy gamma-ray telescopes. *Astroparticle Physics*, 36(1):156–165, 2012.

[22] K. Bestuzheva et al. The SCIP Optimization Suite 8.0. Technical report, Optimization Online, December 2021.

[23] P. N. Bhat, C. A. Meegan, A. von Kienlin, et al. The third Fermi GBM gamma-ray burst catalog: The first six years. *The Astrophysical Journal Supplement Series*, 223(2):28, Apr. 2016.

[24] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Syst.*, 30(1–2):129–154, May 2005.

[25] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser. Timing analysis of a protected operating system kernel. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 339–348, 2011.

[26] T. Blass, A. Hamann, R. Lange, D. Ziegenbein, and B. B. Brandenburg. Automatic Latency Management for ROS 2: Benefits, Challenges, and Open Problems. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 264–277, 2021.

[27] A. Block, B. Brandenburg, J. H. Anderson, and S. Quint. An Adaptive Framework for Multiprocessor Real-Time Systems. In *2008 Euromicro Conference on Real-Time Systems*, pages 23–33, 2008.

[28] S. Boggs and P. Jean. Event reconstruction in high resolution Compton telescopes. *Astronomy and Astrophys. Supp. Series*, 145(2):311–321, 2000.

[29] V. Brocal, P. Balbastre, R. Ballester, and I. Ripoll. Task period selection to minimize hyperperiod. In *ETFA2011*, pages 1–4, 2011.

[30] J. Buckley, S. Alnussirat, C. Altomare, R. G. Bose, D. L. Braun, J. H. Buckley, J. Buhler, E. Burns, R. D. Chamberlain, W. Chen, M. L. Cherry, L. Di Venere, J. Dumonthier, M. Errando, S. Funk, F. Giordano, J. Hoffman, Z. Hughes, D. J. Huth, P. L. Kelly, J. F. Krizmanic, M. Kuwahara, F. Licciulli, G. Liu, M. N. Mazziotta, J. G. Mitchell, J. W. Mitchell, G. A. de Nolfo, R. Paoletti, R. Pillera, B. F. Rauch, D. Serini, G. E. Simburger, M. Sudvarg, G. Suarez, T. Tatoli, G. S. Varner, E. A. Wulf, A. Zink, and W. V. Zober. The Advanced Particle-astrophysics Telescope (APT) Project Status. In *Proc. of 37th International Cosmic Ray Conference — PoS(ICRC2021)*, volume 395, pages 655:1–655:9, July 2021.

[31] G. Bunting, P. Lindsay, A. Maghareh, A. Prakash, and S. Dyke. Using multi-time stepping in finite element models to meet real time constraints. In *EMI/PMC 2012 Joint Conference of the Engineering Mechanics Institute and the 11th ASCE Joint Specialty Conference on Probabilistic Mechanics and Structural Reliability*, 2012.

[32] A. Burns and R. I. Davis. A survey of research into mixed criticality systems. *ACM Comput. Surv.*, 50(6), 11 2017.

[33] M. Burri, J. Nikolic, P. Gohl, T. Schneider, J. Rehder, S. Omari, M. W. Achtelik, and R. Siegwart. The EuRoC micro aerial vehicle datasets. *The International Journal of Robotics Research*, 35(10):1157–1163, 2016.

[34] G. Buttazzo and L. Abeni. Adaptive rate control through elastic scheduling. In *Proceedings of the 39th IEEE Conference on Decision and Control (Cat. No.00CH37187)*, volume 5, pages 4883–4888 vol.5, 2000.

[35] G. Buttazzo and L. Abeni. Adaptive Workload Management through Elastic Scheduling. *Real-Time Systems*, 23(1):7–24, Jul 2002.

[36] G. Buttazzo, M. Velasco, and P. Marti. Quality-of-Control Management in Overloaded Real-Time Systems. *IEEE Transactions on Computers*, 56(2):253–266, 2007.

[37] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, chapter Handling Overload Conditions, pages 287–347. Springer US, New York, 3rd edition, 2011.

[38] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, chapter Resource Access Protocols, pages 205–248. Springer US, New York, 3rd edition, 2011.

[39] G. C. Buttazzo, G. Lipari, and L. Abeni. Elastic Task Model for Adaptive Rate Control. In *IEEE Real-Time Systems Symposium*, 1998.

[40] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic Scheduling for Flexible Workload Management. *IEEE Transactions on Computers*, 51(3):289–302, Mar. 2002.

[41] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. LITMUSˆRT: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 111–126, 2006.

[42] C. Campos, R. Elvira, J. J. G. Rodríguez, J. M. M. Montiel, and J. D. Tardós. ORB-SLAM3: An Accurate Open-Source Library for Visual, Visual–Inertial, and Multimap SLAM. *IEEE Transactions on Robotics*, 37(6):1874–1890, 2021.

[43] L.-C. Canon, M. E. Sayah, and P.-C. Héam. A comparison of random task graph generation methods for scheduling problems. In R. Yahyapour, editor, *Euro-Par 2019: Parallel Processing*, pages 61–73, Cham, 2019. Springer International Publishing.

[44] T. Chantem, X. S. Hu, and M. D. Lemmon. Generalized Elastic Scheduling. In *IEEE International Real-Time Systems Symposium*, 2006.

[45] T. Chantem, X. S. Hu, and M. D. Lemmon. Generalized Elastic Scheduling for Real-Time Tasks. *IEEE Transactions on Computers*, 58(4):480–495, April 2009.

[46] J. Chen. Partitioned multiprocessor fixed-priority scheduling of sporadic real-time tasks. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 251–261, Los Alamitos, CA, USA, jul 2016. IEEE Computer Society.

[47] W. Chen, J. Buckley, et al. Simulation of the instrument performance of the Antarctic Demonstrator for the Advanced Particle-astrophysics Telescope in the presence of the MeV background. In *Proc. of 38th Int'l Cosmic Ray Conference*, volume 444, pages 841:1–841:9. Sissa Medialab, July 2023.

[48] W. Chen, J. H. Buckley, S. Alnussirat, C. Altomare, R. G. Bose, D. L. Braun, J. Buhler, E. Burns, R. D. Chamberlain, M. L. Cherry, L. Di Venere, J. Dumonthier, M. Errando, S. Funk, F. Giordano, J. Hoffman, Z. Hughes, D. J. Huth, P. L. Kelly, J. F. Krizmanic, M. Kuwahara, F. Licciulli, G. Liu, M. N. Mazziotta, J. G. Mitchell, J. W. Mitchell, G. A. de Nolfo, R. Paoletti, R. Pillera, B. F. Rauch, D. Serini, G. E. Simburger, M. Sudvarg, G. Suarez, T. Tatoli, G. S. Varner, E. A. Wulf, A. Zink, and W. V. Zober. The Advanced Particle-astrophysics Telescope: Simulation of the Instrument Performance for Gamma-Ray Detection. In *Proc. of 37th International Cosmic Ray Conference — PoS(ICRC2021)*, volume 395, pages 590:1–590:9, July 2021.

[49] J. W. Condori Uribe, M. Salmeron, E. Patino, H. Montoya, S. J. Dyke, C. E. Silva, A. Maghareh, M. Najarian, and A. Montoya. Experimental benchmark control problem for multi-axial real-time hybrid simulation. *Frontiers in Built Environment*, 9, 2023.

[50] V. Connaughton et al. Localization of gamma-ray bursts using the Fermi gamma-ray burst monitor. *The Astrophysical Journal Supplement Series*, 216(2):32, Feb. 2015.

[51] J. Corbet. Deadline scheduling for linux. *LWN.net*, 2009.

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

[53] S. K. Dhall. *Scheduling periodic-time-critical jobs on single processor and multiprocessor computing systems.* University of Illinois at Urbana-Champaign, 1977.

[54] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations research*, 26(1):127–140, 1978.

[55] S. Dinh, C. Gill, and K. Agrawal. Efficient Deterministic Federated Scheduling for Parallel Real-Time Tasks. In *Proc. of IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2020.

[56] C. Dürr, T. Erlebach, N. Megow, and J. Meißner. Scheduling with explorable uncertainty. In *9th Innovations in Theoretical Computer Science Conference (ITCS 2018)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2018.

[57] A. Easwaran, I. Lee, O. Sokolsky, and S. Vestal. A Compositional Scheduling Framework for Digital Avionics Systems. In *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 371–380, 2009.

[58] F. Eisenbrand and T. Rothvoss. Static-priority real-time scheduling: Response time computation is NP-hard. In *Proceedings of the Real-Time Systems Symposium*, Barcelona, December 2008. IEEE Computer Society Press.

[59] F. Eisenbrand and T. Rothvoß. EDF-schedulability of synchronous periodic task systems is coNP-hard. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, January 2010.

[60] G. A. Elliott, K. Yang, and J. H. Anderson. Supporting Real-Time Computer Vision Workloads Using OpenVX on Multicore+GPU Platforms. In *2015 IEEE Real-Time Systems Symposium*, pages 273–284, 2015.

[61] P. Emberson, R. Stafford, and R. Davis. Techniques for the synthesis of multiprocessor tasksets. In *WATERS workshop at the Euromicro Conference on Real-Time Systems*, pages 6–11, July 2010. 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems ; Conference date: 06-07-2010.

[62] M. Fan and G. Quan. Harmonic semi-partitioned scheduling for fixed-priority real-time tasks on multi-core platform. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 503–508, 2012.

[63] D. Ferry, G. Bunting, A. Maqhareh, A. Prakash, S. Dyke, K. Aqrawal, C. Gill, and C. Lu. Real-time system support for hybrid structural simulation. In *2014 International Conference on Embedded Software (EMSOFT)*, pages 1–10, Oct 2014.

[64] D. Ferry, A. Maghareh, G. Bunting, A. Prakash, K. Agrawal, C. Gill, C. Lu, and S. Dyke. On the performance of a highly parallelizable concurrency platform for real-time hybrid simulation. In *The Sixth World Conference on Structural Control and Monitoring*, 2014.

[65] C. Gill, J. Loyall, R. Schantz, M. Atighetch, J. Gossett, D. Gorman, and D. Schmidt. Integrated adaptive QoS management in middleware: a case study. In *Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium, 2004.*, pages 276–285, 2004.

[66] C. Gill, J. Orr, and S. Harris. Supporting Graceful Degradation through Elasticity in Mixed-Criticality Federated Scheduling. In *6th International Workshop on Mixed Criticality Systems (WMC) at RTSS*, 12 2018.

[67] C. D. Gill, J. M. Gossett, D. Corman, J. P. Loyall, R. E. Schantz, M. Atighetchi, and D. C. Schmidt. Integrated Adaptive QoS Management in Middleware: A Case Study. *Real-Time Systems*, 29(2):101–130, 3 2005.

[68] I. Gog, S. Kalra, P. Schafhalter, J. E. Gonzalez, and I. Stoica. D3: A dynamic deadline-driven approach for building autonomous vehicles. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 453–471, New York, NY, USA, 2022. Association for Computing Machinery.

[69] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2):187–205, Sep 2003.

[70] D. Griffin, I. Bate, and R. I. Davis. Generating Utilization Vectors for the Systematic Evaluation of Schedulability Tests. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 76–88, 2020.

[71] D. Gruber, A. Goldstein, V. W. von Ahlefeld, et al. The Fermi GBM gamma-ray burst spectral catalog: Four years of data. *The Astrophysical Journal Supplement Series*, 211(1):12, Feb. 2014.

[72] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024.

[73] C.-C. Han and K.-J. Lin. Scheduling distance-constrained real-time tasks. In *[1992] Proceedings Real-Time Systems Symposium*, pages 300–308, 1992.

[74] C.-C. Han and H.-Y. Tyan. A better polynomial-time schedulability test for real-time fixed-priority scheduling algorithms. In *Proceedings Real-Time Systems Symposium*, pages 36–45, 1997.

[75] Y. Htet, M. Sudvarg, J. Buhler, R. Chamberlain, W. Chen, J. H. Buckley, et al. Prompt and Accurate GRB Source Localization Aboard the Advanced Particle Astrophysics Telescope (APT) and its Antarctic Demonstrator (ADAPT). In *Proc. of 38th Int'l Cosmic Ray Conference*, volume 444, pages 956:1–956:9. Sissa Medialab, July 2023.

[76] Y. Htet, M. Sudvarg, J. Buhler, R. D. Chamberlain, and J. H. Buckley. Localization of gamma-ray bursts in a balloon-borne telescope. In *Proc. of Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W)*, pages 395–398. ACM, Nov. 2023.

[77] J. R. Jackson. Scheduling a production line to minimize maximum tardiness. *Management Science Research Project*, 43, 1955.

[78] X. Jiang, N. Guan, X. Long, and W. Yi. Semi-Federated Scheduling of Parallel Real-Time Tasks on Multiprocessors. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 80–91. IEEE, 2017.

[79] M. Jones. What really happened on Mars Rover Pathfinder. *The Risks Digest*, 19(49):1–2, 1997.

[80] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, 01 1986.

[81] H. Kellerer, U. Pferschy, and D. Pisinger. *The Multiple-Choice Knapsack Problem*, pages 317–347. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[82] J. Kim, H. Kim, K. Lakshmanan, and R. Rajkumar. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *2013 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, pages 31–40, 2013.

[83] H. Kopetz. On the design of distributed time-triggered embedded systems. *Journal of Computing Science and Engineering*, 2(4):340–356, 2008.

[84] T.-W. Kuo and A. Mok. Load adjustment in adaptive real-time systems. In *[1991] Proceedings Twelfth Real-Time Systems Symposium*, pages 160–170, 1991.

[85] I. Kuz, Y. Liu, I. Gorton, and G. Heiser. Camkes: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software*, 80(5):687––699, May 2007.

[86] W. Kywe, D. Fujiwara, and K. Murakami. Scheduling of Image Processing Using Anytime Algorithm for Real-time System. In *Proc. of 18th Int'l Conf. on Pattern Recognition*, volume 3, pages 1095–1098, 2006.

[87] A. H. Lang, S. Vora, H. Caesar, L. Zhou, J. Yang, and O. Beijbom. PointPillars: Fast Encoders for Object Detection From Point Clouds. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12689–12697, 6 2019.

[88] J. Leonard and H. Durrant-Whyte. Simultaneous map building and localization for an autonomous mobile robot. In *Proceedings IROS '91:IEEE/RSJ International Workshop on Intelligent Robots and Systems '91*, pages 1442–1447 vol.3, 1991.

[89] H. Leontyev and J. H. Anderson. A Hierarchical Multiprocessor Bandwidth Reservation Scheme with Timing Guarantees. In *2008 Euromicro Conference on Real-Time Systems*, pages 191–200, 2008.

[90] A. Li, H. Liu, J. Wang, and N. Zhang. From timing variations to performance degradation: Understanding and mitigating the impact of software execution timing in slam. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2022.

[91] A. Li, J. Wang, S. Baruah, B. Sinopoli, and N. Zhang. An Empirical Study of Performance Interference: Timing Violation Patterns and Impacts. In *2024 Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2024.

[92] H. Li, J. Sweeney, K. Ramamritham, R. Grupen, and P. Shenoy. Real-time support for mobile robotics. In *The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings.*, pages 10–18, 2003.

[93] J. Li, K. Agrawal, C. Lu, and C. Gill. Analysis of Global EDF for Parallel Tasks. In *2013 25th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–13, Los Alamitos, CA, USA, jul 2013. IEEE Computer Society.

[94] J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 85–96. IEEE, 2014.

[95] J. Li, D. Ferry, S. Ahuja, K. Agrawal, C. Gill, and C. Lu. Mixed-Criticality Federated Scheduling for Parallel Real-Time Tasks. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, 2016.

[96] J. Li, D. Ferry, S. Ahuja, K. Agrawal, C. Gill, and C. Lu. Mixed-criticality federated scheduling for parallel real-time tasks. *Real-Time Systems*, 53(5):760–811, 9 2017.

[97] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[98] J. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise computations. *Proceedings of the IEEE*, 82(1):83–94, 1994.

[99] J. M. López, J. L. Díaz, and D. F. García. Utilization Bounds for EDF Scheduling on Real-Time Multiprocessor Systems. *Real-Time Systems*, 28(1):39–68, Oct 2004.

[100] C. Lu, X. Wang, and X. Koutsoukos. End-to-end utilization control in distributed real-time systems. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, pages 456–466, 2004.

[101] C. Lu, X. Wang, and X. Koutsoukos. Feedback utilization control in distributed real-time systems with end-to-end tasks. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):550–561, 2005.

[102] A. Lyons, K. McLeod, H. Almatary, and G. Heiser. Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[103] K. Y. Ma, P. Chirarattananon, S. B. Fuller, and R. J. Wood. Controlled Flight of a Biologically Inspired, Insect-Scale Robot. *Science*, 340(6132):603–607, 2013.

[104] M. Marinoni and G. Buttazzo. Adaptive DVS management through elastic scheduling. In *2005 IEEE Conference on Emerging Technologies and Factory Automation*, volume 2, pages 7 pp.–313, 2005.

[105] S. Mars. Gurobi 10.0.3 released. Technical report, Gurobi Optimization, September 2023.

[106] P. Mészáros, D. B. Fox, C. Hanna, and K. Murase. Multi-messenger astrophysics. *Nature Reviews Physics*, 1(10):585–599, 2019.

[107] N. Min-Allah, I. Ali, J. Xing, and Y. Wang. Utilization bound for periodic task set with composite deadline. *Computers & Electrical Engineering*, 36(6):1101–1109, 2010.

[108] M. Mohaqeqi, M. Nasri, Y. Xu, A. Cervin, and K.-E. Årzén. Optimal harmonic period assignment: complexity results and approximation algorithms. *Real-Time Systems*, 54(4):830–860, Oct 2018.

[109] P. L. Montgomery. A survey of modern integer factorization algorithms. *CWI quarterly*, 7(4):337–366, 1994.

[110] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardós. ORB-SLAM: A Versatile and Accurate Monocular SLAM System. *IEEE Transactions on Robotics*, 31(5):1147–1163, 2015.

[111] M. Nasri and G. Fohler. An Efficient Method for Assigning Harmonic Periods to Hard Real-Time Tasks with Period Ranges. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 149–159, 2015.

[112] M. Nasri, G. Fohler, and M. Kargahi. A Framework to Construct Customized Harmonic Periods for Real-Time Systems. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 211–220, 2014.

[113] L. Nava, G. Ghirlanda, G. Ghisellini, and A. Celotti. Spectral properties of 438 GRBs detected by Fermi GBM. *Astronomy & Astrophysics*, 530:A21, Apr. 2011.

[114] A. Neronov. Introduction to multi-messenger astronomy. In *Journal of Physics: Conference Series*, volume 1263. IOP Publishing, 2019.

[115] S. Odagiri and H. Goto. On the greatest number of paths and maximal paths for a class of directed acyclic graphs. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 97(6):1370–1374, 2014.

[116] U. of Minnesota. U of M leading $1 million grant to build superfast 'TURBO' telescopes, March 2024.

[117] N. A. of Sciences Engineering and Medicine. *Pathways to Discovery in Astronomy and Astrophysics for the 2020s*. The National Academies Press, Washington, DC, 2023.

[118] J. Orr and S. Baruah. Multiprocessor scheduling of elastic tasks. In *Proc. of 27th International Conference on Real-Time Networks and Systems*, pages 133–142. ACM, 2019.

[119] J. Orr, C. Gill, K. Agrawal, S. Baruah, et al. Elasticity of workloads and periods of parallel real-time tasks. In *Proc. of 26th International Conference on Real-Time Networks and Systems*, pages 61–71. ACM, 2018.

[120] J. Orr, C. Gill, K. Agrawal, J. Li, and S. Baruah. Elastic Scheduling for Parallel Real-Time Systems. *Leibniz Transactions on Embedded Systems*, 6(1):05:1–05:14, May 2019.

[121] J. Orr, J. C. Uribe, C. Gill, S. Baruah, et al. Elastic scheduling of parallel real-time tasks with discrete utilizations. In *Proc. of 28th International Conference on Real-Time Networks and Systems*, pages 117–127. ACM, 2020.

[122] Overview of the Fermi GBM. https://fermi.gsfc.nasa.gov/ssc/data/analysis/documentation/Cicerone/Cicerone_Introduction/GBM_overview.html, Jan. 2020. Curated by J.D. Meyers. Accessed: 26 Oct, 2022.

[123] I. Pavić and H. Džapo. Optimal Harmonic Period Assignment With Constrained Number of Distinct Period Values. *IEEE Access*, 8:175697–175712, 2020.

[124] P. Pedreiras and L. Almeida. The flexible time-triggered (FTT) paradigm: an approach to QoS management in distributed real-time systems. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 9 pp.–, 2003.

[125] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, et al. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[126] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for QoS management. In *Proceedings Real-Time Systems Symposium*, pages 298–307, 1997.

[127] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. Practical solutions for QoS-based resource allocation problems. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, pages 296–306, 1998.

[128] J. Real and A. Crespo. Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal. *Real-Time Systems*, 26(2):161–197, 3 2004.

[129] J. Regehr and U. Duongsaa. Preventing interrupt overload. *ACM SIGPLAN Notices*, 40(7):50–58, 2005.

[130] I. Ripoll and R. Ballester-Ripoll. Period Selection for Minimal Hyperperiod in Periodic Task Systems. *IEEE Transactions on Computers*, 62(9):1813–1822, 2013.

[131] P. W. A. Roming, T. E. Kennedy, K. O. Mason, et al. The Swift Ultra-Violet/Optical Telescope. *Space Science Reviews*, 120(3):95–142, Oct. 2005.

[132] S. M. Salman, S. Mubeen, F. Marković, A. V. Papadopoulos, and T. Nolte. Scheduling Elastic Applications in Compositional Real-Time Systems. In *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2021.

[133] S. M. Salman, A. V. Papadopoulos, S. Mubeen, and T. Nolte. Multi-processor scheduling of elastic applications in compositional real-time systems. *Journal of Systems Architecture*, 122:102358, 2022.

[134] The seL4 Microkernel. https://docs.sel4.systems/projects/sel4/. Accessed: 23 Jan, 2022.

[135] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[136] C.-S. Shih, S. Gopalakrishnan, P. Ganti, M. Caccamo, and L. Sha. Scheduling real-time dwells using tasks with synthetic periods. In *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*, pages 210–219, 2003.

[137] I. Shin and I. Lee. Compositional real-time scheduling framework. In *25th IEEE International Real-Time Systems Symposium*, pages 57–67, 2004.

[138] I. Shin and I. Lee. Compositional real-time scheduling framework with periodic model. *ACM Trans. Embed. Comput. Syst.*, 7(3), may 2008.

[139] J. Solem. The application of microrobotics in warfare. Technical report, Los Alamos National Lab, United States, 1996. Research Org.: Los Alamos National Lab. (LANL), Los Alamos, NM (United States); Sponsor Org.: USDOE, Washington, DC (United States); Report Number: LA-UR-96-3067; Contract Number: W-7405-ENG-36; Availability: OSTI as DE96014737.

[140] A. Soyyigit, S. Yao, and H. Yun. Anytime-Lidar: Deadline-aware 3D Object Detection. In *2022 IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 31–40, Los Alamitos, CA, USA, aug 2022. IEEE Computer Society.

[141] M. Stigge, P. Ekberg, and W. Yi. The fork-join real-time task model. *SIGBED Rev.*, 10(2):20, jul 2013.

[142] H. Su and D. Zhu. An Elastic Mixed-Criticality Task Model and Its Scheduling Algorithm. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 147–152, 2013.

[143] M. Sudvarg, S. Baruah, and C. Gill. Elastic Scheduling for Fixed-Priority Constrained-Deadline Tasks. In *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 11–20, 2023.

[144] M. Sudvarg, J. Buhler, J. H. Buckley, W. Chen, et al. A Fast GRB Source Localization Pipeline for the Advanced Particle-astrophysics Telescope. In *Proc. of 37th International Cosmic Ray Conference — PoS(ICRC2021)*, volume 395, pages 588:1–588:9, 7 2021.

[145] M. Sudvarg, J. Buhler, R. Chamberlain, C. Gill, and J. Buckley. Work in Progress: Real-Time GRB Localization for the Advanced Particle-astrophysics Telescope. In *Proc. of 15th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, pages 57–61, 7 2022.

[146] M. Sudvarg, J. Buhler, R. D. Chamberlain, C. Gill, J. Buckley, and W. Chen. Parameterized workload adaptation for fork-join tasks with dynamic workloads and deadlines. In *Proc. of IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 232–242. IEEE, Aug. 2023.

[147] M. Sudvarg et al. Front-End Computational Modeling and Design for the Antarctic Demonstrator for the Advanced Particle-astrophysics Telescope. In *Proc. of 38th International Cosmic Ray Conference*, volume 444, pages 764:1–764:9. Sissa Medialab, July 2023.

[148] M. Sudvarg and C. Gill. A Concurrency Framework for Priority-Aware Intercomponent Requests in CAmkES on seL4. In *2022 IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2022.

[149] M. Sudvarg and C. Gill. Analysis of federated scheduling for integer-valued workloads. In *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, RTNS 2022, page 12–23, New York, NY, USA, 2022. Association for Computing Machinery.

[150] M. Sudvarg, C. Gill, and S. Baruah. Linear-time admission control for elastic scheduling. *Real-Time Systems*, 57(4):485–490, 10 2021.

[151] M. Sudvarg, A. Li, D. Wang, S. Baruah, J. Buhler, C. Gill, N. Zhang, and P. Ekberg. Elastic Scheduling for Harmonic Task Systems. In *2024 Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2024.

[152] M. Sudvarg, Z. Sun, A. Li, C. Gill, and N. Zhang. Priority-based concurrency and shared resource access mechanisms for nested intercomponent requests in CAmkES. *Real-Time Systems*, Apr 2024.

[153] M. Sudvarg, J. Wheelock, J. D. Buhler, J. H. Buckley, and W. Chen. Parallel GRB Source Localization Pipelines for the Advanced Particle-Astrophysics Telescope. In *Proc. of IEEE/ACM International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, Nov. 2021.

[154] M. Sudvarg, C. Zhao, Y. Htet, M. Konst, T. Lang, N. Song, R. D. Chamberlain, J. Buhler, and J. H. Buckley. Hls taking flight: Toward using high-level synthesis techniques in a space-borne instrument. In *Proc. of 21st International Conference on Computing Frontiers*. ACM, 2024.

[155] Y.-C. Tian and L. Gui. QoC elastic scheduling for real-time control systems. *Real-Time Systems*, 47(6):534–561, Dec 2011.

[156] T. Tidwell, R. Glaubius, C. D. Gill, and W. D. Smart. Optimizing Expected Time Utility in Cyber-Physical Systems Schedulers. In *2010 31st IEEE Real-Time Systems Symposium*, pages 193–201, 2010.

[157] P. Turán. On an extremal problem in graph theory. *Matematikai és Fizikai Lapok*, 48:436–452, 1941.

[158] N. Ueter, G. Von Der Brüggen, J.-J. Chen, J. Li, and K. Agrawal. Reservation-Based Federated Scheduling for Parallel Real-Time Tasks. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 482–494. IEEE, 2018.

[159] J. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975.

[160] S. Venugopalan and O. Sinnen. Ilp formulations for optimal task scheduling with communication delays on parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 26(1):142–151, 2015.

[161] S. Vestal. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 239–243, 2007.

[162] A. von Kienlin, C. A. Meegan, W. S. Paciesas, et al. The second Fermi GBM gamma-ray burst catalog: The first four years. *The Astrophysical Journal Supplement Series*, 211(1):13, Feb. 2014.

[163] A. von Kienlin, C. A. Meegan, W. S. Paciesas, et al. The fourth Fermi-GBM gamma-ray burst catalog: A decade of data. *The Astrophysical Journal*, 893(1):46, Apr. 2020.

[164] C. Wang, C. Gill, and C. Lu. FRAME: Fault Tolerant and Real-Time Messaging for Edge Computing. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 976–985, 2019.

[165] C. Wang, C. Gill, and C. Lu. Adaptive Data Replication in Real-Time Reliable Edge Computing for Internet of Things. In *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 128–134, 2020.

[166] D. Wang, J. Zhang, J. Buhler, and J. Wang. Real-time analysis of aerosol size distributions with the fast integrated mobility spectrometer (FIMS). In *41st Conference of American Association for Aerosol Research (AAAR)*, Oct. 2023.

[167] J. Wang, M. Pikridas, S. R. Spielman, and T. Pinterich. A fast integrated mobility spectrometer for rapid measurement of sub-micrometer aerosol size distribution, Part I: Design and model evaluation. *Journal of Aerosol Science*, 108:44–55, 2017.

[168] Q. Wang and G. Parmer. FJOS: Practical, predictable, and efficient system support for fork/join parallelism. In *Proc. of IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 25–36, 2014.

[169] Y. Wang, T. Pinterich, and J. Wang. Rapid measurement of sub-micrometer aerosol size distribution using a fast integrated mobility spectrometer. *Journal of Aerosol Science*, 121:12–20, 2018.

[170] S. Wasly and R. Pellizzoni. Bundled Scheduling of Parallel Real-Time Tasks. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 130–142, 2019.

[171] J. Wheelock, W. Kanu, M. Sudvarg, et al. Supporting Multi-messenger Astrophysics with Fast Gamma-ray Burst Localization. In *Proc. of IEEE/ACM HPC for Urgent Decision Making Workshop (UrgentHPC)*, 11 2021.

Appendix A

Pathological Task Set for Heuristic Partitioned EDF Compression

In Section 3.4 of this dissertation, we consider approaches to find the minimum amount of compression λ (see Equation 3.1) to apply to a task system Γ to achieve partitioned EDF schedulability. Iterative and binary search approaches are presented that find a value for λ within some tunable distance ϵ of the optimal value λ^* if an optimal partitioning is used. However, if heuristic partitioning schemes are employed, it is possible that the values of λ found by each search technique differ by more than ϵ. Table A.1 provides the parameters for a set of 32 tasks for which this occurs when compressed to be scheduled on 8 cores.

Task τ_i Index	$U_i^{\min}$	$U_i^{\max}$	E_i
1	0.06985516371992277	0.1408770839901014	0.041285043560771464
2	0.18570636151919112	0.11070306726529903	0.03130254947936972
3	0.2525641196240303	0.01894732005762579	0.04707499614885927
4	0.1236106771543786	0.1760188035202266	0.19000422938137626
5	0.010518460459193196	0.3139399721904668	0.18719441977559542
6	0.41661454986809876	0.028056400667300172	0.7118021323197531
7	0.1867182811075279	0.02998940800806894	0.044995850922467204
8	0.10864739090063034	0.5031388395951516	0.22373309762411334
9	0.8692788741181185	0.5962645274261063	0.23319608415308588
10	0.2508822172348113	0.36455920461238295	0.20137734477560723
11	0.30400878907117523	0.22235523162679527	0.12305193425185863
12	0.5565126622579573	0.07284690757025704	0.3590808244804951
13	0.2540966927652533	0.4581309185354437	0.3740055387481975
14	0.4643518902030421	0.5692809752286381	0.6229276557843765
15	0.43523325679790087	0.22780548579221138	0.6708578861546909
16	0.3176264661081393	0.22327633447276524	0.9397691384052386
17	0.564158564887814	0.9393469077978592	0.39057363646359
18	0.045310166819438336	0.1644928032402111	0.3115399973977264
19	0.6921248380837226	0.5933718583868578	0.9608689023281628
20	0.8739504009401796	0.4190958795164858	0.3327270729680689
21	0.3786011925228112	0.8963885586264199	0.4087385109668603
22	0.5598561414973221	1.9313164879412992	4.195255260527544
23	4.934119546182837	2.019605371602338	2.466242737779707
24	2.907900281114892	1.554939308858022	3.8884137447724587
25	1.2399943581507458	4.229388837610564	2.0224813853201598
26	3.12756026209997	1.225738825784005	4.898388914956815
27	4.4907850395122075	4.738628641773596	1.8223601409856705
28	1.5199186047637916	4.393753845411336	2.101517563375857
29	4.642178098011004	2.44727587839706	4.557431038205934
30	1.9184983732709031	4.549337521973085	4.438795409642444
31	3.590610938087055	2.8385658033790597	1.7204632772820347
32	1.3648449128066598	3.9435528345317756	1.3618427581680894

Table A.1: Pathological task set parameters