Summer 8-15-2019

Period and Computational Elasticity for Adaptive Real-Time Systems

James Wiliam Orr
Washington University in St. Louis

WASHINGTON UNIVERSITY IN ST.LOUIS

School of Engineering & Applied Science
Department of Computer Science and Engineering

Dissertation Examination Committee:
Christopher D. Gill, Chair
Kunal Agrawal
Sanjoy Baruah
Roger Chamberlain
Shirley Dyke

Period and Computational Elasticity for Adaptive Real-Time Systems

by
James W. Orr

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

August 2019
St. Louis, Missouri

Table of Contents

List of Figures

List of Tables

Acknowledgments

First, and most importantly, thank you to my Savior Jesus Christ. Through him all things are possible.

I would like to thank Chris Gill for his guidance and direction throughout my time here. Without his expertise, patience, understanding, and willingness to take me under his wing, I would not be where I am today. I could not have asked for a better academic advisor.

Kunal Agrawal and Sanjoy Baruah have also been invaluable faculty mentors in my time at Washington University. Their input and advice have been critical to my research career, particularly with theoretical aspects of this dissertation.

Special thanks to Chenyang Lu, Ron Cytron, and Iain Bate for their vital contributions to my graduate career as well. Each contributed valuable time and effort in helping me succeed. Jing Li developed the federated scheduling paradigm upon which my theoretical contributions are based. David Ferry's foundational implementation of concurrency platforms for parallel real-time systems provided a basis I could extend in developing the systems described in this paper. Both of them were immensely helpful in helping this work come to fruition.

Thanks to Shirley Dyke and Arun Prakash in the Department of Civil Engineering at Purdue University and their students Greg Bunting, Amin Maghareh, and Johnny Condori Uribe for their collaboration on work regarding real-time hybrid simulation.

I would also like to thank my undergraduate researcher collaborators Chris Wong, Christian Cianfarani, Phyllis Ang, and Sabina Adhikari for their help and contributions, particularly in helping run experiments.

Thanks also to the National Science Foundation which funded my graduate career under awards CNS-1060093, CCF-113073, CNS-1329861, CCF-1337218, CNS-1814739, and CNS-1911460.

Finally, I would like to thank my wonderful, beautiful wife Taylor for her constant love and support. Without her encouragement none of this would have been possible. Thanks to my daughter Hazel and unborn son for making me the happiest and luckiest Daddy in the world.

James W. Orr

Washington University in Saint Louis
August 2019

Dedicated to Taylor, my constant source of inspiration and encouragement.

ABSTRACT OF THE DISSERTATION

Period and Computational Elasticity for Adaptive Real-Time Systems

by

James W. Orr

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2019

Professor Christopher D. Gill, Chair

A wide range range of real-world applications (including multimedia players, ad-hoc communication networks, online trading, radar tracking software, and other adaptive control algorithms) need adaptive adjustment to their resource utilizations at run-time, while still maintaining real-time guarantees. The elastic task model of soft real-time systems allows for the run-time manipulation of tasks' processor utilizations in order to maintain a system-wide quality of service or accommodate needs of other tasks by assigning each task a period within a specified range. As originally presented, only sequential tasks executing on a single processor were considered. However, in the two decades since the elastic task model was first introduced, multiprocessor systems have become increasingly prevalent. This dissertation appropriately extends the elastic task model to include both multiprocessor scheduling of sequential adaptive tasks and scheduling of adaptive tasks with internal parallelism. It also introduces novel elastic concepts in which 1) tasks can vary their computational loads rather than their periods and 2) the more realistic scenario in which tasks are allowed to adapt among a discrete set of candidate processor utilizations rather than over a continuous range. A runtime system for parallel elastic tasks is also presented and used to demonstrate the benefit of discrete elastic scheduling by enabling adaptation in the application domain of real-time hybrid simulation (RTHS).

Chapter 1

Introduction

The focus of this dissertation lies at the intersection of multi-core real-time scheduling and adaptive real-time scheduling. Specifically we extend the ***elastic task model*** of adaptive real-time tasks from sequential tasks running on a single preemptive processor to include both inter-task and intra-task parallelism running on preemptive multi-core systems. This chapter more precisely details the problem statement, defines the dissertation's specific research contributions, and outlines the remainder of the dissertation.

1.1 Problem Statement and Context

Traditionally real-time systems have been static systems, largely designed and implemented on special-purpose embedded hardware with limited computational power and memory bandwidth, usually running on a single processor. For decades the modeling of these systems' tasks has largely followed the Liu and Layland recurrent task model [37]. Under this model tasks are abstracted to:

- a pessimistic ***worst-case execution time (WCET)*** value which represents an upper-bound on the time it takes to complete a single job of a task on the given processor
- a ***minimum inter-arrival time (or period)*** value which represents the least amount of time between successive jobs of a task
- a ***relative deadline*** (which may be equal to the period) which indicates by how long after the task is released it must finish execution.

Liu and Layland also introduced schedulability tests based on tasks' processor ***utilization***, or the fraction of a processor a task needs in order to guarantee completion by its deadline. If system-wide utilization is below a certain value (depending on the scheduling algorithm used), such schedulability tests can guarantee that a task set will never miss a deadline so long as no task overruns its WCET.

Because of the safety-critical nature of real-time application domains such as the avionic and space industries, in which lives and millions of dollars of equipment potentially may be lost due to deadline misses, excessive pessimism must be used when establishing WCET values, often at the expense of over-provisioning for the average case. Even if a branch of code will almost certainly never run under normal operating modes, the WCET value used to determine schedulability must incorporate the possibility that it will. Task sets for these ***hard real-time systems*** must be certified that they will never miss a deadline. This requires meticulous engineering and extensive design at both hardware and software levels in addition to extreme validation testing.

In contrast, ***soft real-time systems*** do not need to be certified to never miss a deadline under any circumstance. Instead, they make a best effort attempt to provide predictable real-time behavior under most conditions. Rather than the extensive testing and certification required for hard real-time systems, soft real-time systems may use the highest observed

running time (potentially with additional padding as a precaution) among several thousand representative iterations as a WCET value. This allows soft real-time systems to potentially have a less pessimistic view of a task set than hard real-time systems when performing schedulability analysis.

However, the Liu and Layland task model may not always be the best task model for soft real-time systems. In any scenario in which a task's period or computational load (or both) may vary over time, the highest possible system utilization must be accounted for in schedulability analysis under the Liu and Layland model, which may further exacerbate pessimism. Therefore, other task models have been introduced for these ***adaptive real-time systems***. Example applications include multimedia systems, control systems, ad-hoc communication networks, online trading, and radar tracking systems, among others [1, 8, 13, 32, 39].

One such task model created for adaptive real-time systems is the ***elastic task model***. The model was first introduced by Buttazzo et al. [7] to allow sequential tasks running on a single processor to adapt their periods in order for the system to remain schedulable in case a new task must be admitted to the system or an adaptive task must run at a different rate. The model uses an extended analogy to compare schedulability of tasks in a task set to a set of springs laid end-to-end with a common force applied to them in order to compress their combined flength to a specified maximum. The utilization of each task becomes the length of its corresponding spring, and the desired system-wide utilization is the target maximum combined length for the set of springs. Just as some springs are easier to compress than others, and each spring has physical bounds on how far it can be compressed or expanded, each task in the elastic scheduling model has an elasticity parameter to indicate how resistant it is to changing its period, and bounds on which values can be selected as its period (and therefore bounds on its utilization) [7].

In recent decades multi-processor systems have become progressively more popular and readily available. As individual processor speeds plateau, parallel and multi-core programming has become a primary means to achieve increased throughput. Real-time systems have likewise increasingly utilized multiple processors, thereby enabling the exploitation of both inter-task and intra-task parallelism. For instance, intra-task parallelism has allowed for previously unachievable combinations of high computational demands and fine-grained time-scales in high-performance parallel real-time applications such as those in autonomous vehicles [31] and real-time hybrid simulation systems [18, 20]. However, current parallel real-time systems usually assign parallel tasks to fixed sets of processors and release them at statically determined periodic rates [18, 19, 31]. Little work has been done with adaptive parallel real-time systems. Therefore, it is fitting that the elastic task model should also be extended to consider multiprocessors.

1.2 Contributions

The primary contributions of this dissertation are the parallel and multi-processor extensions provided to the elastic model of real-time tasks first introduced by Buttazzo et al. in [7] and the new modes of adaptation made available to adaptive real-time systems on multiprocessors in doing so. Specifically, we make the following contributions:

1. We introduce internal task parallelism to the elastic task model via the federated scheduling paradigm for parallel systems.

2. We further extend the elastic task model to include scheduling of sequential tasks on multiple processors.

3. We extend the notion of task elasticity beyond period adaptation to include computational workload adaptation.

4. We extend the notion of task elasticity to allow for a discrete set of candidate period and computational workload combinations rather than continuous ranges of them.

5. We have developed a run time system for parallel real-time elastic tasks which is used to implement the first adaptive virtual real-time hybrid simulation experiment.

1.3 Organization

The remainder of this dissertation is structured as follows. Chapter 2 provides context for this work by providing relevant background information. We first discuss the original elastic task model and prior works that have extended it. We then discuss the scheduling paradigms used in this dissertation to extend elastic scheduling. We first discuss federated scheduling of parallel real-time tasks. We then discuss both global and partitioned multi-core scheduling of sequential real-time tasks.

Chapter 3 introduces the scheduling of parallel real-time elastic tasks under the federated scheduling paradigm. We first introduce the parallel elastic task model. We then introduce two period-selection and core-allocation algorithms to schedule these tasks under federated scheduling. In our first proposed algorithm we attempt to remain true to the semantics of the uniprocessor elastic task model as proposed by Buttazzo et al. by "stretching" each task equally. However, this algorithm proves to potentially under-utilizize the system under federated scheduling, so we introduce a second algorithm in which tasks are scheduled by attempting to minimize an objective function weighted by their elastic coefficients. This algorithm increases system resource utilization at the cost of some semantic preservation. We then prove this algorithm to optimally solve the associated objective function. This work was published in the Leibniz Transactions on Embedded Systems (LITES) journal in May 2019 [43].

Chapter 4 examines the scheduling of sequential elastic tasks on multiple identical processors. We first review the elastic task model and introduce an algorithm for task period-selection for multi-core elastic scheduling that maintains the semantics of the one proposed by Buttazzo et al.: all task periods are "stretched" equally from their minimum period as weighted by their elastic coefficients. We then study the effects of scheduling tasks via a global vs partitioned manner by generating thousands of task sets and simulating various partitioned and global schedules for them. We make recommendations based upon our findings. This work is to appear at the 27th International Conference on Real-Time Networks and Systems (RTNS) in November 2019 [41].

Chapter 5 extends the notion of task elasticity to include computational elasticity in which a task's computational load can vary rather than its period. It also introduces a run time system for parallel elastic tasks, which we use to demonstrate the functional equivalence between period and computational elasticity. This work was published at the 26th International Conference on Real-Time Networks and Systems (RTNS) in October 2018 [42].

Chapter 6 introduces discrete elastic scheduling. Under this concept, each task has a discrete set of period and workload values rather than a continuous range of one or the other. We discuss the benefits of discrete elastic scheduling which include enabling individual tasks to simultaneously utilize both period and computational elasticity. We also demonstrates discrete elasticity's usefulness by enabling the first adaptive virtual real-time hybrid simulation experiment on the parallel elastic runtime system introduced in Chapter 5. This work is in the process of being submitted for publication. Chapter 7 concludes the dissertation.

Chapter 2

Background

In this dissertation, we extend the definition and applicability of real-time elastic scheduling to multi-core and parallel real-time systems. We start out in this chapter by providing some background on the elastic task model, the federated paradigm of parallel real-time scheduling, the global and partitioned paradigms of scheduling sequential tasks on multiprocessor platforms. Doing so enables us to establish a baseline common among all elastic task model extensions presented in subsequent chapters.

2.1 The Elastic Task Model

The *elastic task model* was first introduced by Buttazzo et al. [7] to allow sequential tasks running on a single processor to adapt their periods (and therefore processor utilizations) in order for the system to remain schedulable in case a new task must be admitted to the system or a dynamic task must run at a faster rate.

The approach is based on a sophisticated analogy between (1) uniprocessor tasks maintaining a collective utilization no greater than a desired utilization U_d (e.g. for schedulability, $U_d = 1.0$ for preemptive EDF scheduling) and (2) a set of springs laid end-to-end being compressed by a collective force until their combined length is at or below a desired maximum length. Just as springs have different resistences to compression, and each spring has physical bounds on how far it can be compressed or expanded, each task in the elastic scheduling model has an elasticity parameter to indicate how resistant it is to changing its period, and bounds on which values can be selected as its period (and therefore bounds on its utilization) [7].

The elastic task model itself is a generalization of Liu and Layland's implicit-deadline sporadic task model [37]. In a set $\Gamma = \tau_1 ... \tau_n$ of sporadic tasks, task $\tau_i = \left\langle C_i, T_i^{(max)}, T_i^{(min)}, E_i \right\rangle$ where C_i represents the task's constant *worst-case execution time (WCET)* and the closed range $[T_i^{(min)}, T_i^{(max)}]$ spans all acceptable period values for a task, where a lower period (and therefore higher utilization) is always preferred. The *current/assigned* period is denoted T_i. A task's *elasticity coefficient* E_i is a measure of how relatively easy or difficult it is to change a task's period, analogous to a spring's stiffness as a measure of its resistance to changing its length: a higher elasticity coefficient indicates a more elastic task, which is more willing to adapt its period. Any task τ_i that should not vary its period (and therefore its utilization) at all can set $T_i^{(min)} = T_i^{(max)}$, and τ_i will act like an ordinary (i.e., not elastic) implicit-deadline sporadic task with WCET C_i and period $T_i^{(min)}$.

In the original elastic scheduling work [7] Buttazzo et al. presented an efficient ($\Theta(n^2)$) iterative algorithm (reproduced later in this dissertation as Algorithm 3 in Chapter 4) for task period selection when the system needed to adapt, which (if possible) finds each task τ_i an appropriate period T_i in a way compliant with spring semantics such that $\sum_i U_i = (C_i/T_i) \leq U_d$ and $T_i^{(min)} \leq T_i \leq T_i^{(max)}$ for all tasks τ_i. The algorithm increases each task's period T_i from $T_i^{(min)}$ proportional to its elasticity coefficient E_i (to a maximum of $T_i^{(max)}$).

It ends either when tasks successfully have been assigned periods such that their combined utilization is less than or equal to U_d, or when each task's period has been stretched to $T_i^{(max)}$ and their combined minimum utilization is still greater than U_d, in which case the task set is declared unschedulable.

Chantem et al. [12, 13] later proved this algorithm to be equivalent to solving the following optimization problem:

$$\textbf{minimize} \sum_{i=1}^{n} \frac{1}{E_i}(U_i^{(max)} - U_i)^2 \tag{2.1}$$

such that:

$$U_i^{(min)} \leq U_i \leq U_i^{(max)} \quad \text{for all } \tau_i, \text{ and}$$

and

$$\sum_{i=1}^{n} U_i \leq U_d.$$

where $U_i^{(max)} = \frac{C_i}{T_i^{(min)}}$ represents the maximum possible utilization of a task obtained from running at period $T_i = T_i^{(min)}$.

The original work involving elastic tasks [7] assumed *implicit deadlines* in which $D_i = T_i$, but theory involving the model has since been expanded to include: *constrained deadlines* in which $D_i \leq T_i$ [13], resource sharing [8], and unknown computational load [10]. Work in Chapter 5 of this dissertation explores a similar (but orthogonal) direction to that in [10] except that we assume a variable, yet known and controlled workload. This dissertation leaves the parallel and multi-core versions of these extensions as future work. Building on the work in this dissertation, recent work by Gill et al., has applied parallel elastic scheduling to mixed-criticality systems [21].

2.2 Federated Scheduling and Parallel Real-Time Tasks Model

Federated scheduling is a parallel real-time scheduling paradigm that was proposed by Li et al. [36] for scheduling collections of recurrent parallel tasks upon multiprocessor platforms, when one or more individual tasks may have a computational requirement that exceeds the capacity of a single processor to entirely accommodate it. Under federated scheduling, such tasks (i.e., those with computational requirement exceeding the capacity of a single processor) are granted exclusive access to a subset of processors; the remaining tasks execute upon a shared pool of processors.

In parallel real-time task systems, the computational requirement of a task τ_i (the generalization of the WCET parameter for sequential tasks) is represented by the following two parameters:

1. The ***work*** parameter C_i denotes the cumulative worst-case execution time of all the parallel branches that are executed across all processors. Note that for deterministic parallelizable code (e.g., as represented in the sporadic DAG tasks model [3]; see [9, Chapter 21] for a textbook description) this is equal to the worst-case execution time of the code on a single processor (ignoring communication overhead from synchronizing processors).

2. The ***span*** parameter L_i denotes the maximum cumulative worst-case execution time of any sequence of precedence-constrained pieces of code. It represents a lower bound on the duration of time the code would take to execute, regardless of the number of processors available.

The span of a program is also called the *critical-path length* of the program, and any end-to-end sequence of precedence-constrained pieces of code with cumulative worst-case execution time equal to the span is a *critical path* through the program.

Algorithms are known for computing the *work* and *span* of a task represented as a DAG, in time linear in the DAG representation. The relevance of these two parameters arises from well-known results in scheduling theory concerning the multiprocessor scheduling of precedence-constrained jobs (i.e., DAGs) to minimize makespan. This problem has long been known to be NP-hard in the strong sense [50]; i.e., computationally highly intractable. However, Graham's *list scheduling* algorithm [23], which constructs a work-conserving schedule by executing at each instant in time an available job, if any are present, upon any available processor, performs fairly well in practice.

An upper bound on the makespan of a schedule generated by list scheduling is easily stated. Given the *work* and *span* of the DAG being scheduled, it has been proved in [23] that the makespan of the schedule for a given DAG upon m processors is guaranteed to be no larger than

$$\frac{work - span}{m} + span \tag{2.2}$$

Thus, a good upper bound on the makespan of the list-scheduling generated schedule for a DAG may be stated in terms of only its work and span parameters. Equivalently, if the DAG represents a real-time piece of code characterized by a relative deadline parameter D, $(\frac{work-span}{m} + span) \leq D$ is a sufficient test for determining whether the code will complete by its deadline upon an m-processor platform.

A parallel task τ_i is considered to be a *high-utilization task* if its *utilization* $U_i = \frac{C_i}{T_i} > 1$ and is considered a *low-utilization task* otherwise. Each high-utilization task τ_i receives m_i dedicated processors on which to run; for implicit-deadlines tasks, we need the resulting

makespan to be less than or equal to $D_i = T_i$; i.e.

$$\begin{aligned} & \frac{C_i - L_i}{m_i} + L_i \leq T_i \\ \Leftrightarrow \quad & \frac{C_i - L_i}{m_i} \leq T_i - L_i \\ \Leftrightarrow \quad & m_i \geq \frac{C_i - L_i}{T_i - L_i} \end{aligned}$$

Under federated scheduling, since the number of processors assigned to each high-utilization task is an integer, we therefore have

$$m_i = \left\lceil \frac{C_i - L_i}{T_i - L_i} \right\rceil . \tag{2.3}$$

Under the original federated scheduling model in [36], low-utilization tasks are treated as sequential tasks and are scheduled using existing mechanisms such as global or partitioned EDF scheduling.

2.3 Multi-core Scheduling of Sequential Tasks

Under the *global* paradigm of multiprocessor scheduling for recurrent tasks, individual tasks are not restricted to executing upon specific processors. Instead, a newly-arrived job of a task may begin execution upon any available processor and a preempted job may resume execution at a later point in time upon any processor, not just the one on which it had been executing prior to preemption. We examine multiple global scheduling algorithms in Section 4.4.

Under the *partitioned* paradigm of multiprocessor scheduling for recurrent tasks, each task is assigned to a processor. Once the partitioning of tasks to processors has been accomplished, each processor independently schedules its allotted tasks using whichever uni-processor

scheduling algorithm is appropriate. The act of partitioning Liu & Layland task systems is known to be equivalent to the bin-packing problem[27, 28], and hence NP-hard in the strong sense. However, several polynomial-time heuristics have been proposed. We discuss them further in Section 4.4.

Chapter 3

Scheduling of Parallel Elastic Tasks

Today's high-performance real-time applications (e.g. real-time hybrid simulation [18, 20]) must often execute upon multiprocessor platforms so as to be able to exploit internal parallelism of these tasks across multiple processors to meet high computational demand. Therefore, the original elastic task model, as well as algorithms that were developed by Buttazzo et al. [7, 8] along with accompanying schedulability analysis and run-time scheduling techniques, need to be appropriately extended in order to be useful for these kinds of high-performance real-time applications. In this chapter, we consider multiprocessor scheduling under the *federated scheduling* paradigm (in which each task whose computational demand exceeds the capacity of a single processor is granted exclusive access to multiple processors); we propose a parallel multiprocessor extension to the elastic task model, and provide appropriate algorithms for federated schedulability analysis and federated scheduling of systems represented using our proposed model.

The *elastic task model* was introduced in [7] with the specific aim of providing dynamic flexibility during run-time. The central idea is that if the overall computational demand of

a system exceeds the capacity of the implementation platform to accommodate it all, then individual tasks' computational demands are reduced and the available platform capacity is allocated in a flexible manner to accommodate these reduced demands. Upon multiprocessor platforms, there are several different interpretations possible, as to what an *elastic* manner of distributing the processors may mean. Our proposed extension aligns with earlier work in the sense that throughout this dissertation we are interpreting the elasticity coefficient parameters according to the semantics assigned to them in the uniprocessor context. We believe that this is a critical issue: the elasticity parameters characterize the relative flexibility –the "hard-real-time"ness– of the tasks, and should bear common interpretation regardless of whether implemented on uni- or multi-processors. We, therefore, believe that the preservation of this interpretation is one of the major benefits of our extended models.

The remainder of this chapter is organized in the following manner. In Section 3.1 we formally define the task model. In Section 3.2 we present a relatively simple and efficient algorithm for scheduling parallel elastic tasks upon multiprocessor platforms, which preserves the semantics that were intended for elastic tasks in the uniprocessor context. We also point out how this simple approach may result in an unnecessary degree of platform resource under-utilization. In Section 3.3 we propose an alternative approach that is able to make more efficient use of the platform to provide a superior scheduling solution, at the cost of not being as faithful to the semantics of elasticity as originally defined for the uniprocessor case. We conclude the chapter in Section 3.4 with a brief summary, and place this work within a larger context of ongoing research efforts towards achieving dynamic flexibility in multiprocessor scheduling of parallelizable workloads.

3.1 Parallel Elastic Task Model

Recall that each elastic task has a range of acceptable periods within the rage $[T_i^{(min)}, T_i^{(max)}]$ and an elasticity coefficient E_i. Because we are now using the federated scheduling paradigm, each task additionally has *work* C_i and *span* L_i parameters to represent its WCET. In this chapter, *we consider only the scheduling of exclusively high-utilization tasks* (i.e., tasks that require more than one processor to meet their deadlines). Scheduling of exclusively low-utilization elastic tasks on multiple cores is the focus of Chapter 4. We therefore do not need to consider them for the remainder of this chapter. We do note that it is possible for some tasks to be either high-utilization or low-utilization depending on the selected period. We refer to these as tasks as *hybrid-utilization*. Formally hybrid-utilization tasks are tasks such that $T^{(min)} \leq C_i \leq T^{(max)}$. We leave period-selection for these tasks (and thereby determining whether these tasks should be treated as high-utilization or low-utilization) as future work. For the sake of completeness in this work we can artificially shorten these tasks' $T^{(max)}$ values to be equal to their C_i values and treat them as high-utilization tasks. Let $U_i^{(max)} = C_i/T_i^{(min)}$ and $U_i^{(min)} = C_i/T_i^{(max)}$ denote the maximum (i.e., desired) and the minimum acceptable utilization for τ_i.

That is, we will consider a system $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n elastic parallel high-utilization tasks that is to be scheduled under federated scheduling upon m processors. In the remainder of this chapter we will often represent a task $\tau_i = (C_i, L_i, U_i^{(max)}, U_i^{(min)}, E_i)$ by its work and span parameters, its maximum and minimum utilizations,[1] and its elasticity coefficient. We will seek to compute m_i, the number of processors that are to be devoted to the exclusive use of task τ_i, for each τ_i such that $\sum_{i=1}^{n} m_i \leq m$.

[1]Note that representing the task by its maximum and minimum utilizations is equivalent to representing it by its minimum and maximum periods, since given C_i, one set of parameters can be derived from the other set.

3.2 A first attempt at elastic scheduling of parallel tasks

It is fairly straightforward to show that the desired elasticity property on the tasks that were defined in the original (uniprocessor) elastic tasks model [7] is that

$$\forall\ i, j, \quad \Big(\frac{U_i^{(max)} - U_i}{E_i}\Big) = \Big(\frac{U_j^{(max)} - U_j}{E_j}\Big) \tag{3.1}$$

That is, the elasticity coefficient E_i of task τ_i is a scaling factor on the amount by which it may have its actual utilization reduced from the desired value of $U_i^{(max)}$.

We use λ to denote the desired equilibrium value for all tasks demonstrated in Expression (3.1); for all tasks $\lambda = \big((U_i^{(max)} - U_i)/E_i\big)$. Expression (3.1) suggests that

$$U_i \leftarrow U_i^{(max)} - \lambda E_i$$

However, we also require $U_i \geq U_i^{(min)}$; hence for a given value of λ we choose

$$U_i(\lambda) \leftarrow \max\Big(U_i^{(max)} - \lambda E_i, U_i^{(min)}\Big) \tag{3.2}$$

Equation (3.2) suggests an algorithm for the federated scheduling of parallel task system $\Gamma = \{\tau_1, \ldots, \tau_n\}$ upon m processors. It is evident from inspection of Equation (3.2) that the "best" schedule — the one that compresses tasks' utilizations the least amount necessary in order to achieve schedulability — is the one for which λ is the smallest. Now for a given value of λ, Algorithm 1 can determine, in time linear in the number of tasks, whether the task system can be scheduled upon the m available processors using federated scheduling.

Algorithm 1 Elastic-1(Γ, m, λ)

▷ Γ is the task system and m the number of processors that are available
▷ λ is the compression factor permitted
$m' \leftarrow 0$ ▷ Number of processors needed
for ($\tau_i \in \Gamma$) **do**
  $U_i = \max\left(U_i^{(max)} - \lambda E_i, U_i^{(min)}\right)$ ▷ See Eqn 3.2
  $T_i = C_i/U_i$
  $m_i = \lceil (C_i - L_i)/(T_i - L_i) \rceil$
  $m' \leftarrow m' + m_i$
end for
if ($m' > m$) **then** ▷ Not enough processors.
  return UNSCHEDULABLE
else
  return $\langle m_1, m_2, \ldots, m_n \rangle$ ▷ τ_i gets m_i processors
end if

Note the value of λ can be bounded to the range of $[0, \phi]$ where $\lambda = 0$ represents all tasks receiving their maximum utilizations and ϕ is the maximum value among all tasks of the equation $\left(\frac{U_i^{(max)} - U_i^{(min)}}{E_i}\right)$. $\lambda = \phi$ thus represents all tasks receiving their minimum utilization. By bounding the potential values of λ, we can use binary search within this range and make repeated calls to Algorithm 1 and thereby determine, to any desired degree of accuracy, the smallest value of λ for which the system is schedulable.

3.2.1 Discussion

Semantics-preservation. Algorithm 1 for the federated scheduling of parallel elastic tasks that we have presented above is *semantics preserving* in the following sense: the assignment of actual period values to the tasks (the T_i's) is done in accordance with Equation (3.2), which is the same manner in which periods are assigned in uniprocessor scheduling of elastic tasks. Hence the system developer who seeks to use our proposed elastic task model to implement flexible parallel tasks upon multiprocessor platforms need not "learn" new (or additional) semantics for the elasticity coefficient: this coefficient means exactly the same

thing in the parallel multiprocessor case as it did in the system designer's previous experiences with sequential uniprocessor tasks (the value of this parameter for each task is a relative measure of its degrees of tolerance to having its period increased and its computational demand thereby reduced).

Run-time platform capacity under-utilization. Despite these advantages, however, one can identify two sources of resource under-utilization by Algorithm 3.2.

- First, observe that the number of processors assigned to a task must be *integral*, and is hence equal to the ceiling of an expression. If the expression $(C_i - L_i)/(T_i - L_i)$, which lies within the ceiling operator ($\lceil \cdot \rceil$) when computing the number of processors assigned to task τ_i, is not itself an integer, then one could further reduce the actual period (the T_i value) that is assigned to the task τ_i and thereby assign τ_i more computational capacity than is afforded by Algorithm 1. However, we do not permit this to happen since the resulting assignment may no longer be semantics-preserving in the sense that different tasks may see a reduction in allocated capacity that is not consistent with their relative elasticity coefficients. This difference between $\lceil (C_i - L_i)/(T_i - L_i) \rceil$ and $(C_i - L_i)/(T_i - L_i)$ is thus "wasted" capacity.
- Second, consider the case with two identical elastic tasks, and an odd number of processors. Semantics-preservation dictates that both tasks be treated in the same manner; however, doing so would correspond to assigning the same number of processors to each task and therefore leaving one processor unused. More generally, Algorithm 1 may leave up to $n - 1$ processors unallocated to n identical tasks.

Thus, the simple semantics-preserving scheme presented in this section may under-utilize platform resources. In Section 3.3 we discuss an alternative scheme that makes more efficient use of platform capacity at the cost of additional complexity in the semantics of elasticity.

3.3 More resource-efficient scheduling

The notion of semantics preservation with uniprocessor elastic task scheduling presented in Section 3.2 is simple and intuitive, and very strong: the elasticity coefficient of a task directly indicates the task's tolerance to having its period parameter increased. However, as we saw, remaining faithful to such a strong notion of semantic equivalence comes at the cost of some computing capacity loss and cannot guarantee full utilization of a platform's computing capacity. We now consider a more generalized interpretation of the semantics of uniprocessor elastic tasks. This interpretation was provided by Chantem et al. [13], who proved that the algorithm of Buttazzo et al. [8] for scheduling sequential elastic tasks upon preemptive uniprocessors is equivalent to solving the following constrained optimization problem:

$$\textbf{minimize} \sum_{i=1}^{n} \frac{1}{E_i}(U_i^{(max)} - U_i)^2 \tag{3.3}$$

such that:

$$U_i^{(min)} \leq U_i \leq U_i^{(max)} \quad \text{for all } \tau_i, \text{ and}$$
$$\sum_{i=1}^{n} U_i \leq U_d$$

where U_d is the desired system utilization. We believe that this is a somewhat less natural interpretation of elasticity in task scheduling than the interpretation considered in Section 3.2: it is perhaps unlikely that a typical system designer is thinking of the elasticity coefficients (the E_i parameters) that they assign to the individual tasks, as coefficients to a quadratic optimization problem. Nevertheless, we adopt this notion of elastic interpretation in this section; for this interpretation, we are able to derive a federated scheduling algorithm that

makes far more efficient use of platform computing capacity than was possible under the earlier more intuitive interpretation considered in Section 3.2.

Note that sequential elastic task scheduling only considers CPU utilization when attempting to schedule tasks on a single processor. Specifically, system-wide utilization $\sum_{i=1}^{n} U_i$ must stay below a desired utilization U_d at all times in order to maintain schedulability. As such, task utilizations are decreased by (when possible) increasing individual task periods in proportion to their fraction of system-wide elasticity until either (1) an acceptable schedule is found such that $\sum_{i=1}^{n} U_i \leq U_d$ or (2) each task τ_i has period $T_i = T_i^{(max)}$ with $\sum_{i=1}^{n} U_i > U_d$. If a schedule cannot be found the task set is declared unschedulable.

In federated scheduling of high-utilization tasks, however, system schedulability is no longer a function only of cumulative utilization but rather whether n tasks can be successfully scheduled on m cores. We now give an algorithm for determining processor allocation and schedulability of a task system that allocates the processors *one at a time* to the tasks: Algorithm 2. Algorithm 2 starts out by determining, for each task τ_i, the minimum number of processors $m_{i_{min}}$ needed to be meet its minimum acceptable computational load (i.e., having $T_i \leftarrow T_i^{(max)}$) in Line 2, and the number $m_{i_{max}}$ needed to meet its desired computational load (i.e., having $T_i \leftarrow T_i^{(min)}$) in Line 3. Since the assigned period T_i satisfies $T_i^{(min)} \leq T_i \leq T_i^{(max)}$, the actual number of CPUs m_i assigned to τ_i is also bounded by $m_{i_{min}} \leq m_i \leq m_{i_{max}}$.

Because of the ceiling function in Equation (6.1), each range of values for T_i maps to a given m_i for each task. In this work we assume that it is beneficial for each task to run as frequently as possible. As such, we assign task τ_i the minimum period T_i available on m_i allocated processors. We denote this period value as $T_{(i,m_i)}$, which is derived directly from Equation (6.1):

$$T_{(i,m_i)} = \frac{C_i - L_i}{m_i} + L_i \tag{3.4}$$

Algorithm 2 Task_compress_par(Γ, m)

for ($\tau_i \in \Gamma$) **do**
  $m_{i_{min}} = \lceil (C_i - L_i)/T_{i_{max}} - L_i) \rceil$ ▷ Minimum number of processors
  $m_{i_{max}} = \lceil (C_i - L_i)/(T_{i_{min}} - L_i) \rceil$ ▷ Maximum number of processors
  $m_i = m_{i_{min}}$
  while $m_i <= m_{i_{max}}$ **do** ▷ Compute the shortest period for τ_i
    ▷ for each possible value of m_i
    $T_{(i,m_i)} = (C_i - L_i)/(m_i) + L_i$ ▷ $T_{(i,m_i)}$ = shortest with m_i processors
    $m_i = m_i + 1$
  end while
  $m_i = m_{i_{min}}$ ▷ Assign minimum number of processors
  $T_i = T_{(i,m_i)}$ ▷ Assign corresponding shortest period
  $m = m - m_{i_{min}}$ ▷ m keeps count of processors remaining
end for
if ($m < 0$) **then** ▷ There weren't enough processors
  return UNSCHEDULABLE
else if ($m == 0$) **then**
  return *processor allocation with m_i values*
end if

The remainder of this pseudocode
allocates processors one at a time

for ($\tau_i \in \Gamma$) **do**
  Determine δ_i, the potential
  decrease to Problem 3.5 for each task
end for

Make a max heap of all tasks, with the δ_i values as the key

while m $>$ 0 and heap not empty **do** ▷ Assign remaining processors
  $\tau_{most} = heap.pop()$ ▷ Task that would most benefit
  $m_{most} = m_{most} + 1$ ▷ Permanently assign processor
  $m = m - 1$
  $T_{most} = T_{(most,m_{most})}$
  if ($m > 0$ and $m_{most} < m_{most_{max}}$) **then** ▷ Able to receive more processors?
    Determine δ_{most}, the potential
    decrease to Problem 3.5 for task τ_{most}
    Reinsert τ_{most} into heap
  end if
end while
return *the processor allocation with m_i values*

All possible values of $T_{(i,m_i)}$ for $m_{i_{min}} \leq m_i \leq m_{i_{max}}$ are computed first and stored in lookup tables. This is accomplished during the while loop (Lines 5–9) in Algorithm 2.

Next (Lines 10–12), each task is assigned the minimum number of processors it needs, and this number of processors is subtracted from m; hence at the end of the loop, m denotes the number of processors remaining for additional assignment (above and beyond the minimum needed per task). If $m < 0$ the system is unschedulable, while if $m = 0$ there is nothing more to be done — the system is schedulable with each task receiving its minimum level of service. These conditions are tested in Lines 14–18 of the pseudocode in Algorithm 2.

If $m > 0$, however, we will individually assign each of these remaining m processors to whichever task would benefit "the most" from receiving it. This is determined in the following manner. Similar to scheduling sequential tasks [13], our goal is to find task utilizations (and therefore periods) that solve the optimization problem:

$$\textbf{minimize} \sum_{i=1}^{n} \frac{1}{E_i}(U_i^{(max)} - U_i)^2 \tag{3.5}$$

such that:

$$U_i^{(min)} \leq U_i \leq U_i^{(max)} \text{ for all } \tau_i, \text{ and}$$
$$\sum_{i=1}^{n} m_i \leq m$$

In allocating each processor we calculate, for each task τ_i, a quantity δ_i which represents the *decrease* in $\frac{1}{E_i}(U_i^{(max)} - U_i)^2$ if the next processor were to be allocated to task τ_i — this is done in Lines 23–26 of Algorithm 2. We then assign the processor to whichever task would see the biggest decrease. (As a consequence, the objective function of optimization problem 3.5 would decrease the most.) To accomplish this efficiently, we

- Place the tasks in a max heap indexed on the value of δ_i (Line 28); and
- while there are unallocated processors and the heap is not empty (checked in Line 30)
  - assign the next processor to the task at the top of the heap (Lines 31–34) and, if this task is eligible to receive more processors (checked in Line 35), recompute δ_i for this task (Line 36) and reinsert into the heap (Line 38).

Run-time complexity. The first for-loop in the algorithm (Lines 1–13 in the pseudocode listing in Algorithm 2) takes $\Theta(m * n)$ time. The for-loop in Lines 23–26 and the making of the max heap (Line 28) each take $\Theta(n)$ time. The running time of the remainder of the algorithm (Lines 30–40) is dominated by the max-heap operations; the overall running time is therefore $\Theta(n * m + m \log n)$.

3.3.1 Proof of Optimality

In this section we prove in Theorem 1 that Algorithm 2 solves the optimization problem given in Equation (3.5) optimally. The optimality of Algorithm 2 then follows from the result of Chantem et al. [13] showing the equivalence of uniprocessor elastic scheduling of sequential tasks with the optimization problem given in Equation (5.1).

The dependency amongst the three results in this section — Lemma 1, Lemma 2, and Theorem 1 — is strictly linear: Lemma 1 is needed to prove Lemma 2, which is needed to prove Theorem 1.

Lemma 1. *The utilization U_i of elastic task τ_i strictly increases towards maximum utilization as the number of processors m_i assigned to it increases.*

Proof. Since $U_i = C_i/T_i$, (and C_i is constant), U_i increases as T_i decreases. By Equation (3.4), $T_i = ((C_i - L_i)/m_i) + L_i$. C_i and L_i are constant for task τ_i. Therefore, T_i strictly decreases as m_i increases. Therefore, an increase of m_i decreases T_i and increases U_i. ■

Lemma 2. *In assigning processors one at a time (in the while loop of Lines 30–40 of Algorithm 2), the consecutive assignment of the $(k+1)$'st and $(k+2)$'nd to the same task τ_i with k currently assigned processors will result in diminishing returns of δ_i, the decrease in $\frac{1}{E_i}(U_i^{(max)} - U_i)^2$ for τ_i. (i.e., the benefit of assigning a processor to a task is never as high as the already-incurred benefit of assigning prior processors.)*

Proof. This is readily observed by algebraic simplification.[2] Let x_k be the value of $\frac{1}{E_i}(U_i^{(max)} - U_{i_k})^2$ where U_{i_k} is the task utilization with k processors. Let x_{k+1} be the value of $\frac{1}{E_i}(U_i^{(max)} - U_{i_{k+1}})^2$ with new utilization $U_{i_{k+1}}$ after assigning processor $k+1$ to τ_i, and similarly let x_{k+2} be the value of $\frac{1}{E_i}(U_i^{(max)} - U_{i_{k+2}})^2$ with new utilization $U_{i_{k+2}}$ after subsequently assigning processor $k+2$ to τ_i. From Lemma 1, we know that $U_{i_k} < U_{i_{k+1}} < U_{i_{k+2}}$.

Define the benefit of adding processor $k+1$ to τ_i as $\delta_{i_{k+1}} = x_k - x_{k+1}$, and the later benefit of assigning processor $k+2$ as $\delta_{i_{k+2}} = x_{k+1} - x_{k+2}$. To prove diminishing returns, we must show that $\delta_{i_{k+1}} > \delta_{i_{k+2}}$.

Note that the math is equivalent, so we temporarily ignore the constant scalar $\frac{1}{E_i}$. Thus, both

$$\delta_{i_{k+1}} = (U_i^{(max)} - U_{i_k})^2 - (U_i^{(max)} - U_{i_{k+1}})^2 \tag{3.6}$$

and

$$\delta_{i_{k+2}} = (U_i^{(max)} - U_{i_{k+1}})^2 - (U_i^{(max)} - U_{i_{k+2}})^2 \tag{3.7}$$

[2]The algebra, while straightforward, is rather tedious and the reader may choose to just skim it at first reading.

are of the form

$$(x-z)^2-(x-y)^2 \tag{3.8}$$

where $x > y > z$. We can therefore say that $z+\alpha = y$ and $y+\beta = x$.

Re-stating Equation (3.8) in terms of z, α, and β, we obtain:

$$(z+\alpha+\beta-z)^2-(z+\alpha+\beta-z-\alpha)^2$$

which simplifies to

$$\alpha^2+2\alpha\beta. \tag{3.9}$$

Therefore, to prove $\delta_{i_{k+1}} > \delta_{i_{k+2}}$, it is sufficient to show that

$$\alpha_{k+1}^2+2\alpha_{k+1}\beta_{k+1} > \alpha_{k+2}^2+2\alpha_{k+2}\beta_{k+2} \tag{3.10}$$

where α_{k+1}, β_{k+1}, α_{k+2}, and β_{k+2} are $(U_{i_{k+1}}-U_{i_k})$, $(U_i^{(max)}-U_{i_{k+1}})$, $(U_{i_{k+2}}-U_{i_{k+1}})$, $(U_i^{(max)}-U_{i_{k+1}})$, respectively. (These values come from the definitions of α and β and the substitutions of x, y, and z in Equation (3.8) into their actual values from Equations 3.6 and 3.7.) Note that as α_{k+1}, β_{k+1}, α_{k+2}, and β_{k+2} are all positive numbers, Equation (3.10) will be satisfied if we can individually prove $\alpha_{k+1} > \alpha_{k+2}$ and $\beta_{k+1} > \beta_{k+2}$, which we now proceed to do.

We first prove $\beta_{k+1} > \beta_{k+2}$, where

$$\beta_{k+1} = (U_i^{(max)} - U_{i_{k+1}}),$$

and

$$\beta_{k+2} = (U_i^{(max)} - U_{i_{k+2}}).$$

We know from above that $U_{i_{k+2}} > U_{i_{k+1}}$. Therefore

$$(U_i^{(max)} - U_{i_{k+1}}) > (U_i^{(max)} - U_{i_{k+2}})$$

and $\beta_{k+1} > \beta_{k+2}$.

We next prove $\alpha_{k+1} > \alpha_{k+2}$. Note that

$$U_i = \frac{C_i}{T_i = \frac{C_i - L_i}{m_i} + L_i} \tag{3.11}$$

Consider Equation (3.11) which shows the complete derivation of a task's utilization as a function of the number of processors assigned to it. By definition, if $\alpha_{k+1} \stackrel{?}{>} \alpha_{k+2}$, [3] then

$$U_{i_{k+1}} - U_{i_k} \stackrel{?}{>} U_{i_{k+2}} - U_{i_{k+1}}.$$

Substituting into Equation (3.11), this becomes

$$\frac{C_i}{\frac{C_i - L_i}{k+1} + L_i} - \frac{C_i}{\frac{C_i - L_i}{k} + L_i} \stackrel{?}{>} \frac{C_i}{\frac{C_i - L_i}{k+2} + L_i} - \frac{C_i}{\frac{C_i - L_i}{k+1} + L_i}.$$

Factoring out a constant C_i and simplifying, we get

$$\frac{k+1}{C_i + kL_i} - \frac{k}{C_i + kL_i - L_i} \stackrel{?}{>} \frac{k+2}{C_i + kL_i + L_i} - \frac{k+1}{C_i + kL_i}.$$

Letting $X = C_i + kL_i$ (to enhance readability), this becomes

$$\frac{k+1}{X} - \frac{k}{X - L_i} \stackrel{?}{>} \frac{k+2}{X + L_i} - \frac{k+1}{X}.$$

[3] We use $\stackrel{?}{>}$ to indicate that the inequality is not yet proved.

We can combine fractions and simplify this further to

$$\frac{-kL_i + X - L_i}{X(X - L_i)} \stackrel{?}{>} \frac{-kL_i + X - L_i}{X(X + L_i)}.$$

Since $-k * L_i + X - L_i = -kL_i + C_i + kL_i - L_i = C_i - L_i > 0$ for high-utilization tasks, we can now factor out $-k * L_i + X - L_i$ from both sides and are left with asking whether

$$\frac{1}{X(X - L_i)} \stackrel{?}{>} \frac{1}{X(X + L_i)}.$$

This is unequivocally true. Hence, we prove that $\alpha_{k+1} > \alpha_{k+2}$. Therefore, Equation (3.10) is satisfied and $\delta_{i_{k+1}} > \delta_{i_{k+2}}$. The Lemma follows. ■

Theorem 1. *Algorithm 2 optimally minimizes the optimization problem given in Equation (3.5).*

Proof. For Algorithm 2 to be non-optimal, there must be some point at which our greedy algorithm and the optimal algorithm diverge. (Algorithm 2 begins optimally with the only valid assignment of processors to tasks when considering only the minimum amount of processors each task can have.) Note that each task's contribution to the sum of Equation (3.5) is independent of other tasks: the value of $\frac{1}{E_i}(U_i^{(max)} - U_i)^2$ for a given task τ_i is independent of how many processors have been assigned to other tasks. Thanks to this property, we need only consider two tasks. Let us suppose, without loss of generality, that at the point of divergence our greedy algorithm assigns the processor to τ_i, while the optimal algorithm would assign the processor to τ_j.

Because the greedy algorithm assigns the processor to τ_i, we know that the added benefit (amount decreased from the sum) is greater than if we had given the processor to τ_j. Hence the current value of the objective function of optimization problem 3.5 the greedy algorithm

is necessarily lower than that of the optimal algorithm upon assignment of the number of processors assigned thus far. By the assumption regarding the non-optimality of our greedy strategy, there must be some point in the future at which the optimal algorithm makes up the difference since the optimal solution to a minimization problem must end with the lowest value for the objective function.

However, we saw in Lemma 2 above that the benefits of assigning a new processor under the greedy Algorithm 2 diminish. At each iteration, the greedy algorithm chooses to assign the processor to the task with the greatest available benefit. Because tasks' benefits are considered independently and do not change regardless of the allocation of CPUs to other tasks, after the greedy algorithm assigns the k'th processor to τ_i, no other task τ_j will have a higher benefit of receiving the $(k+1)$'st processor than it did when the greedy algorithm elected to give the k'th processor to τ_i. Similarly, by Lemma 2 the diminishing returns of assigning multiple processors to the same task guarantees that the benefit of assigning the $(k+1)$'st task to τ_i is also less than the benefit gotten by assigning the k'th processor to τ_i. Therefore, if the optimal algorithm and the greedy algorithm diverge and the current value of the objective function of optimization problem 3.5 for Algorithm 2 is better than the optimal algorithm, it is impossible for the optimal algorithm to subsequently "catch up" and do better than the greedy algorithm. Hence the current value of the objective function of optimization problem 3.5 may never diverge between an optimal algorithm and our greedy algorithm; the optimality of Algorithm 2 immediately follows. ■

This completes the proof of optimality of Algorithm 2 for the federated scheduling of parallel elastic tasks.

3.4 Summary

In the two decades since it was first introduced, the elastic task model [7] has proved a useful abstraction for representing flexibility in the computational demands of recurrent workloads. It was originally proposed for representing sequential tasks executing upon uniprocessor platforms; as high-performance real-time computer applications are increasingly becoming parallelizable (and need to have their parallelism exploited by being implemented upon multiprocessor platforms in order to meet timing constraints), there is a need to extend the applicability of the elastic task model to parallel tasks that execute upon multiprocessor platforms.

In this chapter, we have proposed one such extension. The salient features of our model are:

- Multiprocessor scheduling under the federated paradigm, in which each task needing more than one processor is assigned exclusive access to all processors upon which it executes. Federated scheduling frameworks generally can be implemented in a more efficient manner than global scheduling (e.g., with less run-time overhead) with only limited loss of schedulability (as measured by speedup bounds of capacity augmentation bounds) [34, 36].
- Representation of a parallel task's workload using just the cumulative workload (its "*work*" parameter) and its critical path length (its "*span*" parameter). Such representation allows for efficient schedulability analysis in the federated scheduling framework, with a bounded loss of schedulability as compared to DAG representations (for which schedulability analysis is strongly NP-hard.
- Retention of the elasticity coefficient parameter that was the main innovation introduced in [7] to capture the flexibility in computational demands.

We have proposed and studied two schemes for assigning processors to tasks in a system of elastic parallel real-time tasks that are to be scheduled upon a given multiprocessor platform under federated scheduling. One of these schemes is completely semantics-preserving with respect to model semantics as introduced in the uniprocessor case [7]; the other allows for some deviation from uniprocessor semantics and thereby is able to better use the computational capabilities of the implementation platform.

Chapter 4

Multiprocessor Scheduling of Sequential Elastic Tasks

4.1 Introduction

Buttazzo et al. introduced the *elastic task model* as a way of modeling recurrent real-time tasks, such as multimedia players or adaptive control systems, whose periods can change depending on the stress on the system [7]. Each task must be assigned a period within its acceptable range such that the overall task set utilization remains below a desired value. To determine the appropriate period value to assign each task, every task also has an *elastic coefficient* which acts as an indicator of the task's resistance to increasing its period from the minimum (and desired) period, analogous to a spring's resistance to being compressed.

In Chapter 3 we extended the elastic task model to include scheduling of tasks with intra-task parallelism on heterogeneous multi-core systems under the federated scheduling paradigm. In this chapter we focus on the scheduling of sequential tasks on homogeneous multi-core

systems. We present algorithms for scheduling systems of such tasks upon a homogeneous multiprocessor platform under both the global and partitioned paradigms of multiprocessor scheduling. We compare the effectiveness of different algorithms via an extensive series of simulation experiments; based upon the outcomes of these simulations, we make some recommendations regarding the choice of algorithms for the multiprocessor scheduling of sequential elastic tasks.

The remainder of this chapter is structured as follows. Section 4.2 presents our task model. Sections 4.3 and 4.4 discuss the global and partitioned scheduling of tasks respectively. Section 4.5 details our experimental evaluation of the different schemes, and Section 4.6 summarizes this chapter's contributions.

4.2 Task Model and Assumptions

Recall that in the model proposed by Buttazzo et al. [7], each elastic task τ_i is characterized by a worst-case execution time (WCET) C_i, a minimum (and preferred) period $T_i^{(min)}$, a maximum acceptable period $T_i^{(max)}$, and an elasticity coefficient E_i. The elasticity coefficient is a measure of a task's resistance to changing its period. A higher elasticity coefficient indicates a more elastic task. In this chapter we seek to schedule a set of n such independent sequential elastic tasks $\Gamma = \tau_1 ... \tau_n$ on m homogeneous processors.

As stated in previous chapters, an elastic elastic task τ_i is characterized by the parameters

$$\tau_i = \left(C_i, T_i^{(max)}, T_i^{(min)}, E_i \right) \ .$$

All the scheduling approaches that we will consider in this chapter have *utilization-based* schedulability conditions: only the utilization parameters of tasks appear in these schedulability conditions. We therefore find it convenient to convert the period parameters of each task (the $T_i^{(min)}$ and $T_i^{(max)}$ parameters) to corresponding utilization parameters $U_i^{(max)}$ and $U_i^{(min)}$ respectively:

$$U_i^{(max)} = C_i/T_i^{(min)}$$
$$U_i^{(min)} = C_i/T_i^{(max)}$$

In the remainder of this chapter, each task τ_i is therefore characterized by the parameters

$$\tau_i = \left(U_i^{(max)}, U_i^{(min)}, E_i\right) \ .$$

Letting U_i denote the actual utilization "allocated" to τ_i, the desired elasticity property defined by Buttazzo et al. (as also stated in the previous chapter) is equivalent to specifying that the amounts by which tasks' utilizations are reduced from their desired maximums be in proportion to their E_i ("elasticity") coefficients:

$$\forall\ i, j, \quad \left(\frac{U_i^{(max)} - U_i}{E_i}\right) = \left(\frac{U_j^{(max)} - U_j}{E_j}\right) \tag{4.1}$$

Letting λ denote the desired equilibrium value such that for all tasks $\lambda = \left((U_i^{(max)} - U_i)/E_i\right)$, Expression 4.1 suggests

$$U_i \leftarrow U_i^{(max)} - \lambda E_i$$

However, we also require $U_i \geq U_i^{(min)}$; hence we choose

$$U_i(\lambda) \leftarrow \max\Big(U_i^{(max)} - \lambda E_i, U_i^{(min)}\Big) \tag{4.2}$$

Note that for a given value of λ, an elastic task $\tau_i = (U_i^{(max)}, U_i^{(min)}, E_i)$ is just a "regular" Liu and Layland task with utilization $U_i(\lambda)$ as given by Expression 4.2 above.

The problem considered. For each of the multiprocessor scheduling strategies we will study in this chapter, the question we ask is: given an n-task system

$$\Gamma = \Big\{\tau_i = (U_i^{(max)}, U_i^{(min)}, E_i)\Big\}_{i=1}^{n}$$

that is to be scheduled upon an m-processor platform, what is the smallest value of λ for which the Liu and Layland task system comprising n tasks with utilizations $U_1(\lambda)$, $U_2(\lambda)$, ..., $U_n(\lambda)$ is successfully schedulable by that particular scheduling strategy?

4.3 Global Scheduling

Under the global paradigm of multiprocessor scheduling for recurrent tasks, individual tasks are not restricted to executing upon specific processors. Instead, a newly-arrived job of a task may begin execution upon any available processor and a preempted job may resume execution at a later point in time upon any processor, not just the one it had been executing upon prior to preemption. We consider three different global scheduling algorithms: fluid (Section 4.3.1), Earliest Deadline First (Section 4.3.2), and an algorithm called PriD [22] that can be thought of as a generalization of EDF (Section 4.3.3).

4.3.1 Fluid Scheduling

The *fluid scheduling* paradigm of multiprocessor real-time scheduling permits that individual tasks be assigned a fraction f, $0 \leq f \leq 1$, of a processor at each instant in time (in contrast to non-fluid schedules, in which each task may execute either upon zero processors or upon a single processor at each instant). Fluid scheduling is a convenient abstraction that considerably simplifies many multiprocessor real-time scheduling problems; techniques are known (see, e.g, [4, 25, 33, 40]) for converting fluid schedules to non-fluid ones for many problems and under a wide range of conditions and circumstances.

Fluid scheduling of Liu and Layland tasks – a review. Consider some Liu and Layland task system Γ, and let U_i denote the utilization of $\tau_i \in \Gamma$. It has been shown [25] that a necessary and sufficient condition for Γ to be fluid-schedulable upon a multiprocessor platform comprising m unit-speed processors is that

$$max_{\tau_i \in \Gamma}\{U_i\} \leq 1 \tag{4.3}$$

and

$$\left(\sum_{\tau_i \in \Gamma} U_i\right) \leq m \ . \tag{4.4}$$

Any task system satisfying Conditions 4.3 and 4.4 can be fluid-scheduled by simply assigning each job of τ_i a fraction U_i of one of the m processors at each instant between its release time and its deadline.

Extension to elastic tasks. In the original elastic scheduling paper [7], Buttazzo et al. present an iterative algorithm called Task_Compress(Γ,U_d) for assigning a period to each task in a system Γ of elastic tasks such that the total system utilization stays below a desired

value U_d — this algorithm is reproduced in this chapter as Algorithm 3. It is evident that

Algorithm 3 Task_Compress(Γ,U_d)

1: $U^{(max)} = \sum_{i=1}^{n} C_i/T_i^{(min)}$
2: $U^{(min)} = \sum_{i=1}^{n} C_i/T_i^{(max)}$
3: **if** $U_d < U^{(min)}$ **then**
4: **return** INFEASIBLE
5: **end if**
6: ok= 0
7: **while** $ok == 0$ **do**
8: $U_f = E_v = 0$
9: **for** *each* τ_i **do**
10: **if** $E_i == 0$ **or** $T_i == T_i^{(max)}$ **then**
11: $U_f = U_f + U_i$
12: **else**
13: $E_v = E_v + E_i$
14: **end if**
15: **end for**
16: ok= 1
17: **for** *each* $\tau_i \in \Gamma_v$ **do**
18: **if** $E_i > 0$ **and** $T_i < T_i^{(max)}$ **then**
19: $U_i = U_i^{(max)} - (U^{(max)} - U_d + U_f) * E_i/E_v$
20: $T_i = C_i/U_i$
21: **if** $T_i > T_i^{(max)}$ **then**
22: $T_i = T_i^{(max)}$
23: ok= 0
24: **end if**
25: **end if**
26: **end for**
27: **end while**
28: **return** FEASIBLE

Algorithm 3 is, in essence, determining the smallest value of λ for which

$$\left(\sum_{i=1}^{n} U_i(\lambda) \right) \leq U_d \ ,$$

where the $U_i(\lambda)$'s are as defined according to Expression 4.2. Observe, too, that Algorithm 3 never *increases* the actual utilization assigned to any any task τ_i to beyond $U_i^{(max)}$ — this follows from the observation that in Line 19, the value assigned to the actual utilization —the parameter U_i— is obtained by *subtracting* a positive quantity from $U_1^{(max)}$. Hence given an elastic task system Γ of sequential tasks that is to be fluid-scheduled upon m unit-speed processors, we can determine the effective utilizations of the individual tasks that satisfy Conditions 4.3 and 4.4, and therefore bear witness to the fluid-schedulability of Γ, by simply calling the procedure Task_Compress(Γ,U_d) of Algorithm 3 with $U_d \leftarrow m$. The instance Γ can then be fluid-scheduled by assigning each job of each $\tau_i \in \Gamma$ a fraction of a processor equal to this effective utilization at each instant between its release date and its deadline.

4.3.2 Global EDF

While the fluid scheduling model is a convenient abstraction for considering multiprocessor scheduling, it is not in general directly implementable. As mentioned above, techniques are known for converting fluid schedules to non-fluid ones under a variety of conditions; however, most such conversions yield schedules with a large number of preemptions and inter-processor migrations. In environments in which there is a considerable overhead associated with each preemption and/or inter-processor migration, this approach of obtaining a fluid schedule and then converting to a non-fluid one may incur unacceptably high overhead costs.

Review of results for Liu and Layland tasks. The global Earliest Deadline First (EDF) scheduling algorithm has the property that the total number of preemptions and inter-processor migrations in a schedule is bounded from above at the number of jobs in the schedule. (This is easily seen by observing that a job may preempt an already-executing one only upon its arrival, if it happens to have an earlier deadline; such preemption may later lead to an inter-processor migration if the preempted job resumes upon a different

processor.) Global EDF may therefore be a more appropriate algorithm to use in environments characterized by significant preemption/migration overhead costs. Goossens et al. showed [22, Theorem 5] that a system Γ of Liu & Layland tasks is scheduled by global EDF to meet all deadlines upon m unit-speed processors if the following condition holds:

$$\sum_{\tau_i \in \Gamma} U_i \leq m - (m-1) \times \max_{\tau_i \in \Gamma} \{U_i\} \tag{4.5}$$

(This condition was also shown [22, Theorem 6] to be tight from a utilization-based perspective: there are systems in which $\left(\sum_{\tau_i \in \Gamma} U_i\right)$ is greater than$(m - (m-1) \times \max_{\tau_i \in \Gamma} \{U_i\})$ by an arbitrarily small amount, upon which global EDF misses deadlines.)

Extension to elastic tasks. Given a system Γ of elastic tasks

$$\Gamma = \left\{ \tau_i = (U_i^{(max)}, U_i^{(min)}, E_i) \right\}_{i=1}^{n}$$

that is to be scheduled upon an m-processor platform, our objective is to find the smallest value of λ such that the Liu & Layland task system with the following utilizations

$$U_i \leftarrow \left\{ \max\left(U_i^{(max)} - \lambda E_i, U_i^{(min)} \right) \right\}_{i=1}^{n} \tag{4.6}$$

is schedulable using global EDF. We have chosen to solve this problem by iterating through the possible values of λ — see Algorithm 4. This algorithm steps through the range $[0, \Phi]$ with a "granularity" ϵ (Line 1 of Algorithm 4), where Φ is the maximum value among all tasks of the equation $\left(\frac{U_i^{(max)} - U_i^{(min)}}{E_i} \right)$. The algorithm seeks the smallest value of λ or which the Liu & Layland task system of Expression 4.6 above is global EDF-schedulable according to Expression 4.5. Once this smallest value of λ is determined and returned by Algorithm 4, we can convert the elastic task system to a regular Liu & Layland task system by computing

the effective utilizations of the tasks according to Expression 4.2, and then use global EDF to schedule the Liu & Layland task system so obtained.

Algorithm 4 Global EDF(Γ,m)

```
1: ε ← 0.05 × Φ                                   ▷ "Granularity" of the test...
2: for λ ← 0 to Φ by ε do
3:     S ← 0.0                                    ▷ Total utilization of compressed tasks
4:     M ← 0.0                                    ▷ Max. utilization amongst compressed tasks
5:     for i ← 1 to |Γ| do
6:         tmp← max(U_i^(max) − λE_i, U_i^(min))
7:         S ← S+ tmp
8:         M ← max(M, tmp)
9:     end for
10:    if (S ≤ m − (m − 1) × M) then
11:                          ▷ By Eqn. 4.5, the compressed tasks are global-EDF schedulable,
12:        return λ
13:    end if
14: end for
15: return (global EDF fails)
```

4.3.3 Algorithm PriD

It was observed [2] that global EDF tends to under-perform when there is even a single task with high utilization. This is easily explained by examining the utilization-based global-EDF schedulability condition of Inequality 4.5: observe the presence of the

$$\left((m-1) \times \max_{\tau_i \in \Gamma} \{U_i\}\right)$$

term on the right-hand side. Since this term is *subtracted* from the total computing capacity of the platform (i.e., m), the consequence is that a capacity of $(m-1)$ times the largest individual utilization becomes unavailable due to the presence of this large-utilization task.

Algorithm PriD (Γ, m)
The Liu & Layland task system $\Gamma = \{\tau_1, \tau_2, \ldots \tau_n\}$ is to be scheduled on m processors
Assume the tasks are indexed according to utilization: $U_i \geq U_{i+1}$ for all i, $1 \leq i < n$
for $i = 1$ **to** m **do**
if $\{\tau_{i+1}, \tau_{i+2}, \ldots, \tau_n\}$ is global-EDF schedulable upon $(m - i)$ processors
then
During run-time $\{\tau_1, \tau_2, \ldots, \tau_i\}$'s jobs will be assigned highest priority and $\{\tau_{i+1}, \tau_{i+2}, \ldots, \tau_n\}$'s job
return success
return failure // *Not schedulable by PriD*

Figure 4.1: Algorithm PriD priority-assignment rule

This phenomenon can be looked upon a consequence of the well-known *Dhall effect* [15, 16] which has been widely studied in multiprocessor real-time scheduling theory. Several results have been obtained within the real-time scheduling theory community for dealing with such utilization loss; below we first review some of these results and then seek to extend their applicability to incorporate elasticity.

Review of results for Liu and Layland tasks. Recall that one major advantage of EDF-generated schedules over those obtained by converting a fluid-based one is the reduced number of preemptions and inter-processor migrations: the total number of preemptions and migrations in an EDF-generated is no greater than the number of jobs that are scheduled. It turns out that this property is in fact enjoyed by an entire class of algorithms: all those in which each job is assigned a single fixed priority and at each instant during run-time the highest-priority jobs that are eligible to execute are selected for execution. Algorithms in this class are referred to as *Fixed Job Priority* (FJP) [11] scheduling algorithms. The algorithm fpEDF was proposed [2] as an FJP algorithm that circumvents the utilization loss caused by the Dhall effect. Under the fpEDF run-time scheduling algorithm, jobs of tasks with utilization > 0.5 are statically assigned highest priority while priorities to jobs of the

remaining tasks are assigned according to EDF. It has been shown [2, Theorem 4] that a task system Γ is scheduled by fpEDF to meet all deadlines upon m unit-speed processors if the following condition holds:

$$\sum_{\tau_i \in \Gamma} U_i \leq \frac{m+1}{2} \tag{4.7}$$

A pragmatic improvement to fpEDF, called Algorithm PriD (for "priority driven") was proposed by Goossens et al. [22] — this is the algorithm that we will be adapting below for elastic tasks. Algorithm PriD is presented in pseudo-code form in Figure 4.1. Algorithm PriD, like fpEDF, seeks to circumvent the Dhall effect by assigning greatest priority to jobs of tasks with high utilization; however, while fpEDF designates all tasks with utilization > 0.5 to be "high-utilization" ones, Algorithm PriD determines which tasks are "high-utilization" based on the characteristics of the task system under consideration. It is shown [22] that Algorithm PriD strictly dominates fpEDF: all instances that are deemed schedulable by fpEDF are also deemed schedulable by PriD while the converse of this statement is not true – there are instances deemed schedulable by Algorithm PriD that will not pass the fpEDF schedulability test of Expression 4.7.

Extension to elastic tasks. Our adaptation of Algorithm PriD to elastic tasks is similar to our adaptation of global EDF: given an instance of elastic tasks

$$\Gamma = \left\{ \tau_i = (U_i^{(max)}, U_i^{(min)}, E_i) \right\}_{i=1}^{n}$$

to be scheduled upon m unit-speed processors, we iterate through possible values of λ between 0 and Φ, seeking the smallest value such that the Liu & Layland task system with utilizations

$$U_i \leftarrow \left\{ \max(U_i^{(max)} - \lambda E_i, U_i^{(min)} \right\}_{i=1}^{n}$$

is deemed schedulable by Algorithm PriD upon m unit-speed processors. (The pseudo-code for this algorithm is very similar to the pseudo-code in Algorithm 4, and hence omitted.)

4.4 Partitioned Scheduling

The partitioned scheduling of Liu & Layland task systems is known to be equivalent to the bin-packing problem[27, 28], and hence NP-hard in the strong sense. Several polynomial-time heuristics have been proposed for solving this problem approximately: most of these heuristic algorithms for partitioning have the following common structure. First, they specify an order in which the tasks are to be considered. Then in considering each task (in the order chosen), they specify the order in which to consider upon which processor to attempt to allocate the task. A task is successfully allocated upon a processor if it is observed to "fit" upon the processor; within the context of the partitioned EDF-scheduling, a task fits on a processor if the task's utilization does not exceed the processor capacity minus the sum of the utilizations of all tasks previously allocated to the processor. The algorithm declares success if all tasks are successfully allocated; otherwise, it declares failure.

Lopez et al. [38] have extensively compared several widely-used heuristic algorithms that fit this overall structure. They define the concept of a *Reasonable Allocation* (RA) partitioning algorithm: an RA algorithm is one that fails to allocate a task to a multiprocessor platform only when the task does not fit into any processor upon the platform. All the heuristic algorithms considered by Lopez et al. [38] are RA ones — indeed, there seems to be no reason why a system designer would ever consider using a non-RA partitioning algorithm. Within the RA algorithms, Lopez et al. [38] compared heuristics that

1. use three different ways for ordering the tasks to consider: arbitrary, in order of increasing utilization, and in order of decreasing utilization; and

2. also use three different heuristics for ordering the processors to consider: “first fit" (assign a task to the first processor upon which it fits), “worst fit" (assign a task to the processor with the maximum remaining capacity), and and “best fit" (assign a task to the processor with the minimum remaining capacity that exceeds the task's utilization).

Extension to elastic tasks. Any of the partitioning heuristics can be adapted for elastic tasks in a manner that is very similar in structure to the manner in which global EDF and PriD were adapted for elastic tasks. That is, given an instance of elastic tasks

$$\Gamma = \left\{ \tau_i = (U_i^{(max)}, U_i^{(min)}, E_i) \right\}_{i=1}^{n}$$

to be scheduled upon m unit-speed processors, we iterate through possible values of λ between 0 and Φ, seeking the smallest value such that the Liu & Layland task system with utilizations

$$U_i \leftarrow \left\{ \max(U_i^{(max)} - \lambda E_i, U_i^{(min)} \right\}_{i=1}^{n}$$

is deemed schedulable by upon m unit-speed processors by the partitioning heuristic. (The pseudo-code for doing so is again very similar to the pseudo-code in Algorithm 4, and hence omitted.)

We note that after partitioning tasks onto processors, it is highly unlikely that all processors are fully utilized (i.e. the assigned utilizations of the partitioned tasks sum to 1.0). The procedure Task_Compress(Γ,U_d) of Buttazzo et al. [7] (reproduced here as Algorithm 3) can be applied to each processor with $U_d \leftarrow 1.0$ to perhaps increase system utilization while still guaranteeing schedulability,

4.5 Simulation Experiments

We have performed a simulation-based comparison of the various algorithms presented in Sections 4.3 and 4.4 for the multiprocessor scheduling of sequential elastic tasks; we report on the findings of this comparison below. We describe the setup for these simulation experiments in Section 4.5.1 and present our findings in Section 4.5.2; based upon these findings, we draw some high-level conclusions in Section 4.5.3.

4.5.1 Experimental Setup

We randomly generate sets of sequential elastic tasks and attempt to schedule them upon a given number of processors m using the different scheduling algorithms – fluid, global EDF, PriD, and partitioned – described in Sections 4.3 and 4.4 above. Specifically,

- We separately consider multiprocessor platforms containing $m = 4, 8$, and 16 identical processors.
- For each of these values for m, we consider task sets with $n = 2 \times m, 2.5 \times m$, $3 \times m$, and $n = 4 \times m$ tasks.
- For each selected combination of values of m and n, we generate task sets in which the maximum utilizations of the tasks (i.e., their $U_i^{(max)}$ parameters) sum to $1.1 \times m, 1.5 \times m$, and $1.9 \times m$.

Hence a total of $3 \times 4 \times 3 = \mathbf{36}$ different combinations of values of m, n, and $\left(\sum_i U_i^{(max)}\right)$ are considered. For each such combination, we generate 1000 task sets in the following manner. We generate the individual $U^{(max)}$ values using the *Randfixedsum* algorithm [17] to provide an unbiased distribution of maximum utilizations. The corresponding individual task

minimum utilization values $U_i^{(min)}$ are uniformly generated over the range $(0, U_i^{(max)})$. In the case that a task set's $U_i^{(min)}$ values sum to more than m (i.e., the task set is not schedulable under fluid scheduling, or therefore, any other scheduling algorithm), we repeatedly generated new $U_i^{(min)}$ values for each task until their sum is sufficiently low. Tasks' elastic coefficients are chosen uniformly randomly over the range [1, 5]. For all algorithms a "granularity" of $\epsilon = \frac{\Phi}{1{,}000}$ was used.

We attempt to schedule each task set generated as described above using the four algorithms discussed in Sections 4.3 and 4.4: fluid, global EDF, PriD, and partitioned. For partitioned, we first sort the tasks in order of decreasing utilization (their $U_i^{(max)}$ parameters), and attempt to assign them to the available processors using the the "first-fit," "worst-fit," and "best-fit" heuristics; We return the first λ value that deems the task set schedulable by any of these heuristics.

4.5.2 Observations

In our experiments, we noted (i) the fraction of task-sets that were determined to be schedulable by each of our four algorithms; and (ii) for those task-sets that were deemed schedulable by all the algorithms, the minimum λ needed to achieve schedulability by each algorithm. Our results are presented in graphical form in Figures 4.2–4.14. In these graphs we show results of both the average minimum normalized λ value ($\frac{\lambda}{\Phi}$–this gives a value on the interval [0, 1] and is needed to compare λ values across task sets) needed to achieve schedulability for a given scheduling algorithm, and the percentage of the 1,000 task sets that each algorithm deemed scheduleable. To ensure a consistent comparison, we only compare lambda values for task sets deemed schedulable by all scheduling algorithms.

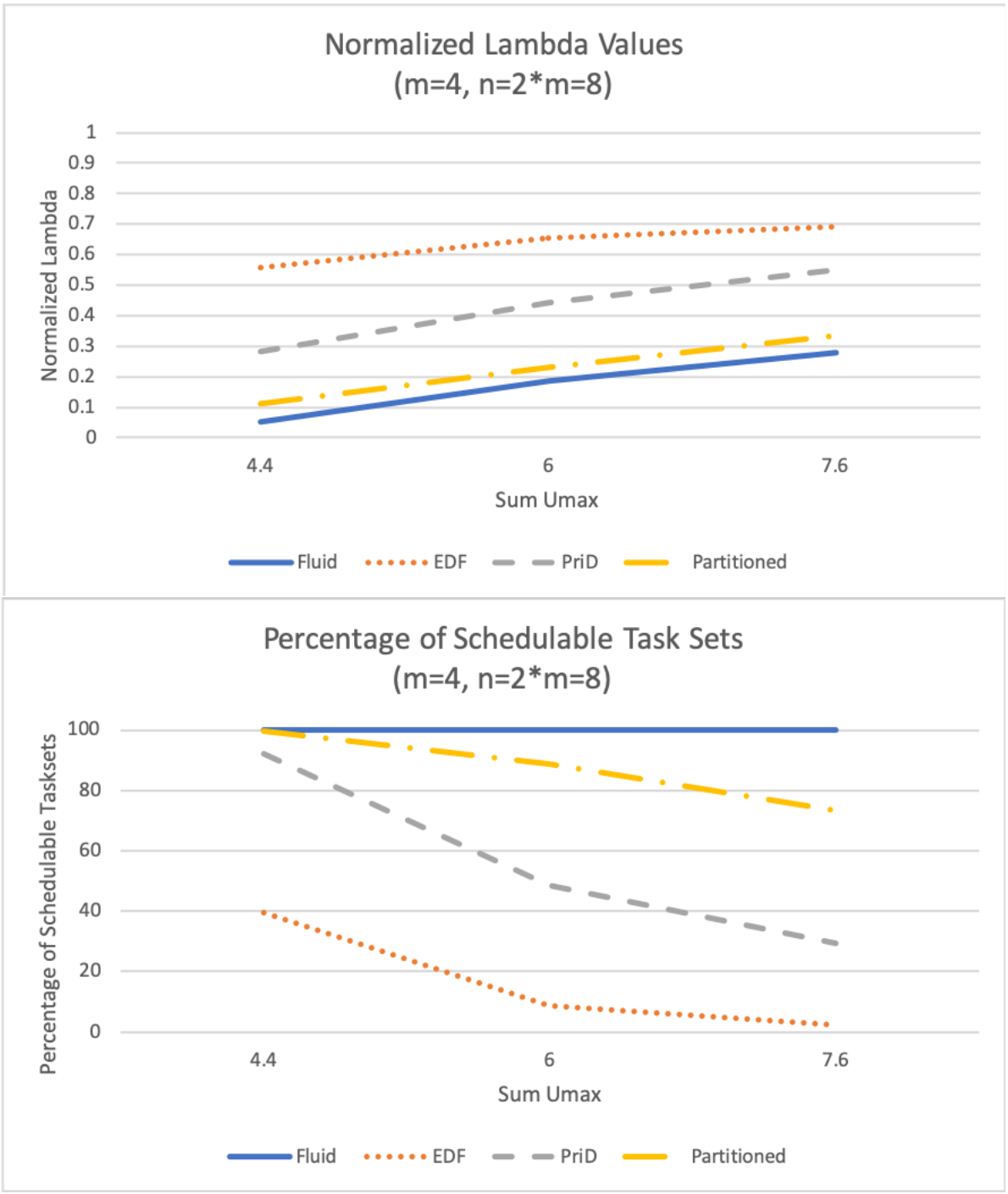


Figure 4.2: All scheduling algorithms considered

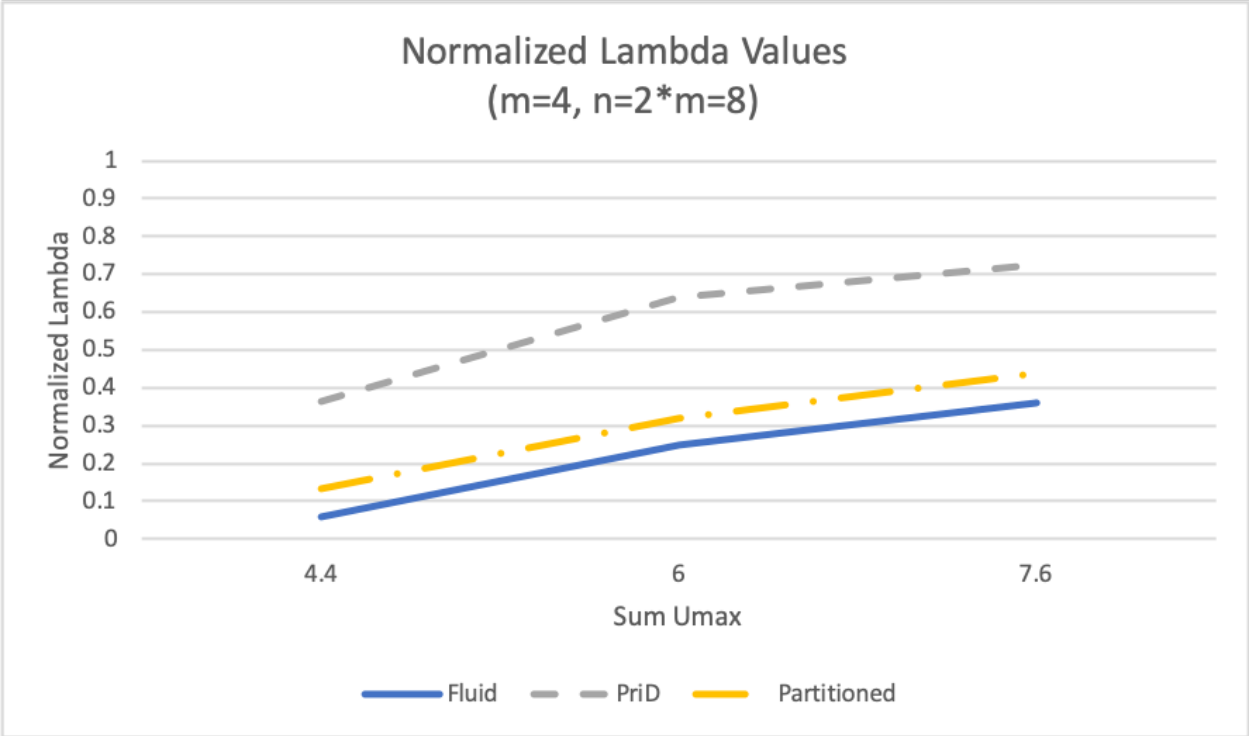
Figure 4.2 shows both the λ values and percentage of schedulable task sets for all four scheduling algorithms (fluid, global EDF, PriD, and partitioned) under our most lax system conditions: $m = 4$ processors and $n = 8$ tasks. Note that global EDF scheduling requires by far the highest lambda value, and that the percentage of task sets deemed schedulable under global EDF falls off drastically as $\left(\sum_i U_i^{(max)}\right)$ increases. This is a manifestation of the Dhall effect, and our further experiments revealed that this only worsens as the number of processors and tasks increase: for many of the other combinations of m, n, and $\left(\sum_i U_i^{(max)}\right)$ that we later considered, global EDF fails to schedule even a single task set out of 1000. Therefore, in order to have more task sets schedulable under "all" scheduling algorithms, we remove global EDF from consideration and only compare fluid, PriD, and partitioned scheduling in the remainder of the reported results.

Figures 4.3 – 4.8 show the average lambda among task sets that were deemed schedulable under all algorithms while Figures 4.9 – 4.14 show the percentage of task sets schedulable under each scheduling algorithm. We note that fluid scheduling is an idealized optimal scheduling algorithm; not surprisingly, therefore, it schedules the largest percentage of task-sets and returns the smallest λ value. This is seen consistently in Figures 4.8 – 4.14. We also note that partitioned scheduling consistently dominates algorithm PriD in both λ value and in percentage of schedulable task sets. This is consistent with prior observations [5] regarding global versus partitioned multiprocessor scheduling; in essence, this is likely a reflection of the fact that while global scheduling algorithms like PriD apply schedulability tests that are utilization-based and incorporate considerable pessimism since they must consider "worst-case" task-sets with the same utilization parameters, partitioned schedulability tests actually attempt to perform a partition and hence do not necessarily pay the price in terms of such analysis-based pessimism.

Our experiments also reveal that it becomes more difficult to schedule tasks (in terms of both λ value and schedulability percentage) for all the scheduling algorithms as $\left(\sum_i U_i^{(max)}\right)$ increases. The same is true as the number of processors increases but the ratio of processors to tasks remains the same. Indeed, no tasks were deemed schedulable under algorithm PriD on $m = 16$ processors, regardless of the ratio of processors to tasks in the system. However, on a constant number of processors, fluid and partitioned scheduling can return a lower λ value with more tasks in the task set, and a higher percentage of task sets are deemed schedulable under partitioned scheduling while PriD seems minimally affected. Fluid scheduling always deems 100% of tasks to be schedulable. We believe this improvement is due to a reduction in the Dhall effect: as more tasks are introduced into a system with a constant total utilization the largest single task is more likely to decrease.

4.5.3 Some Conclusions

Based on our observations in the previous subsection and the graphs in Figures 4.8 – 4.14, we recommend that in the absence of specific knowledge regarding task characteristics that may advocate in favor of PriD, partitioned scheduling be used for the scheduling of sequential elastic tasks on uniform multiprocessor systems, particularly in systems with a large number of tasks. Among the realistic scheduling algorithms considered in this chapter, it 1) consistently returns the lowest value of λ (and therefore compresses tasks the least) and 2) schedules the highest percentage of task sets.



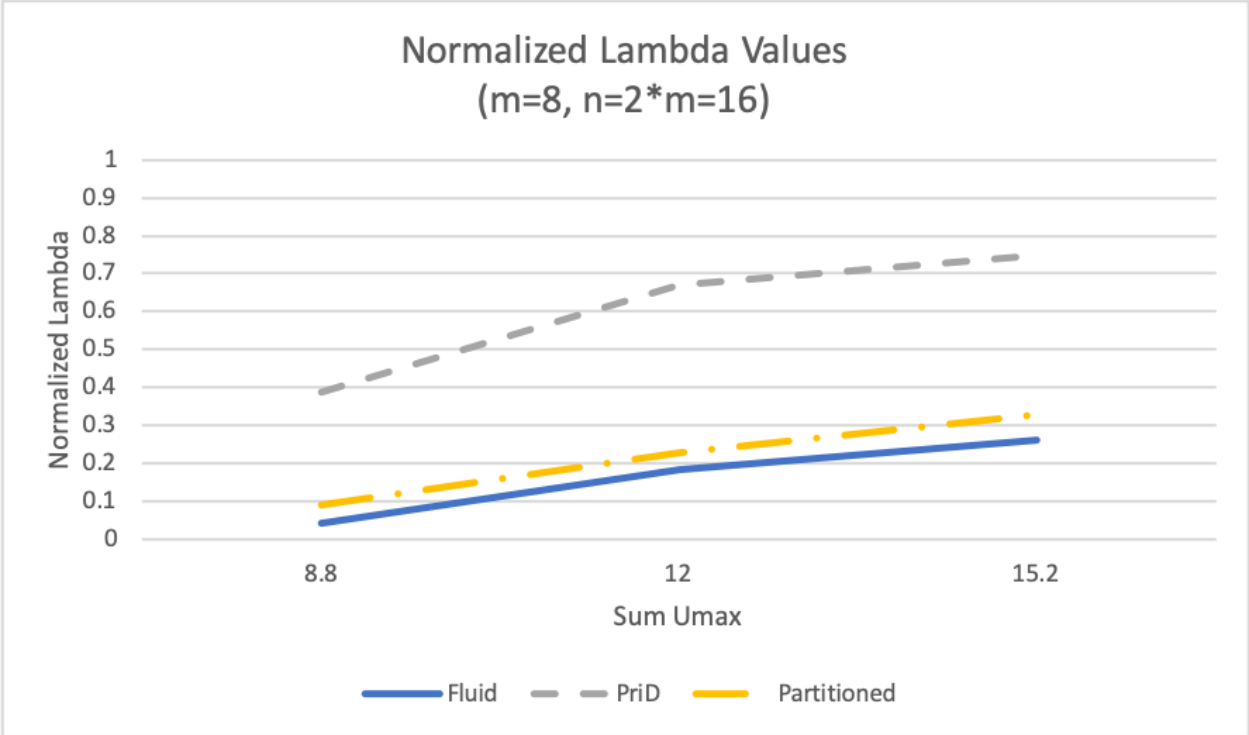
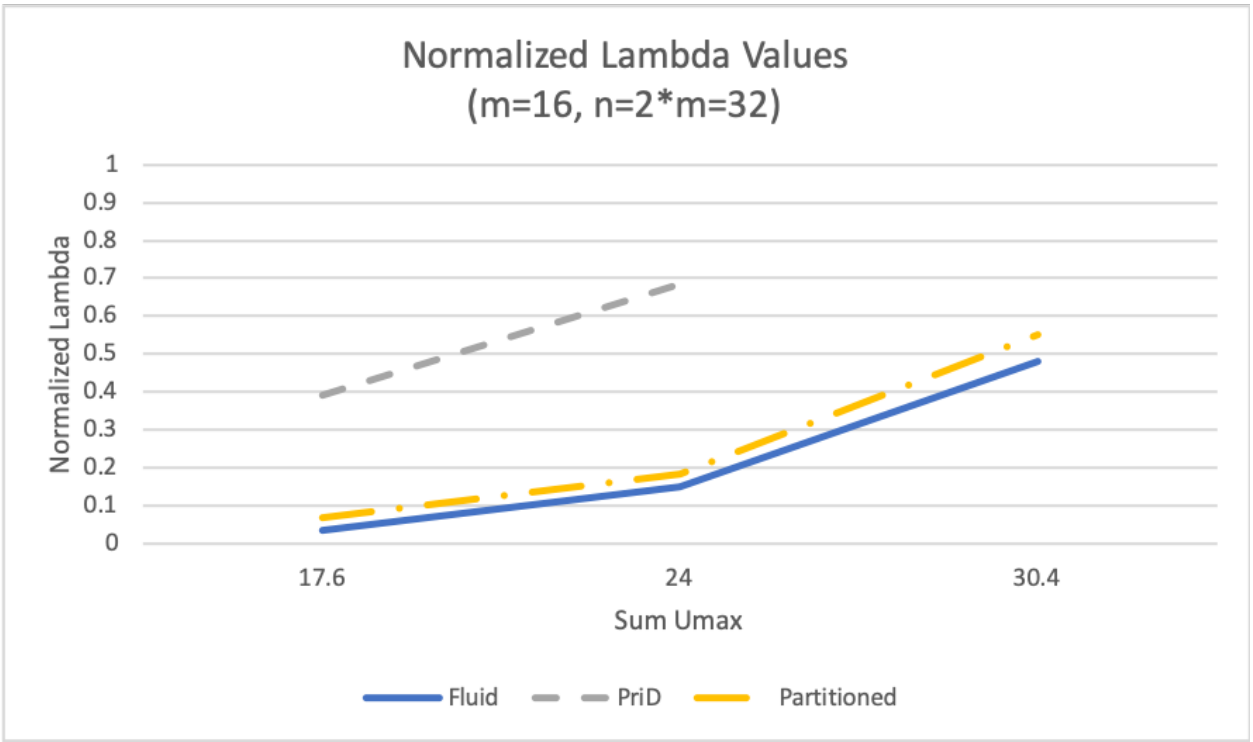


Figure 4.3: Normalized Lambda Values



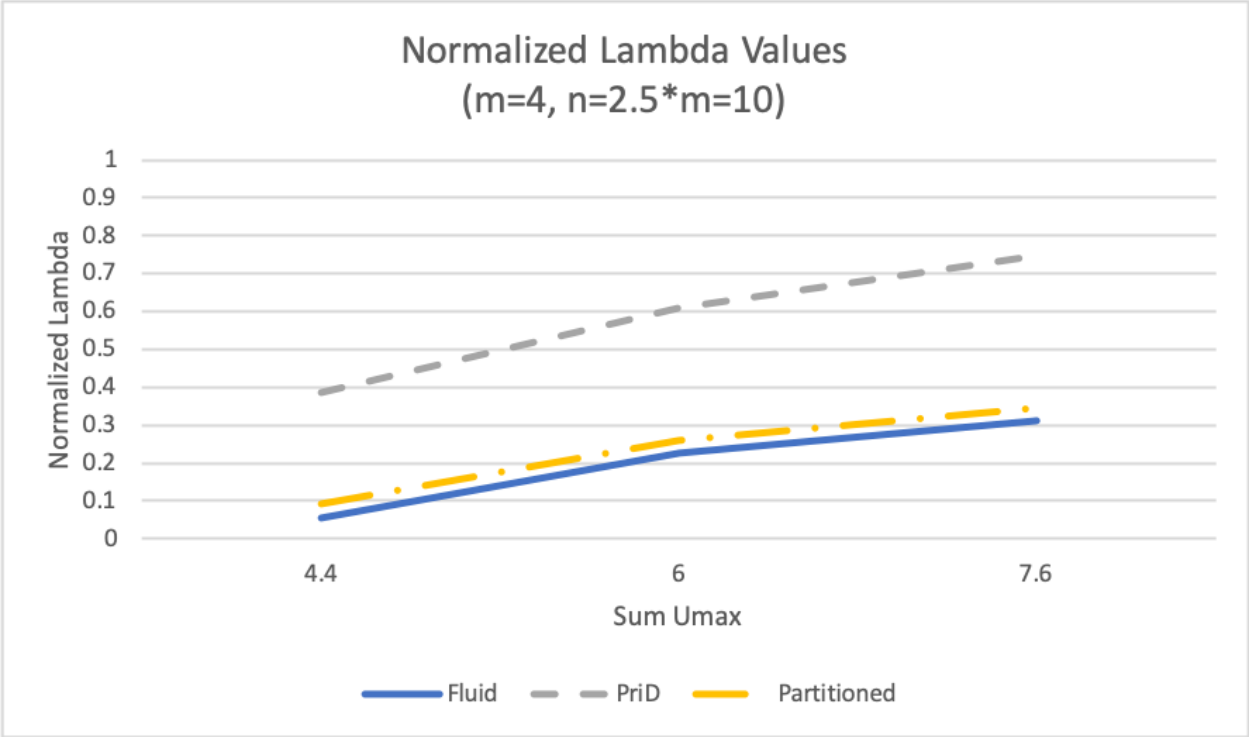
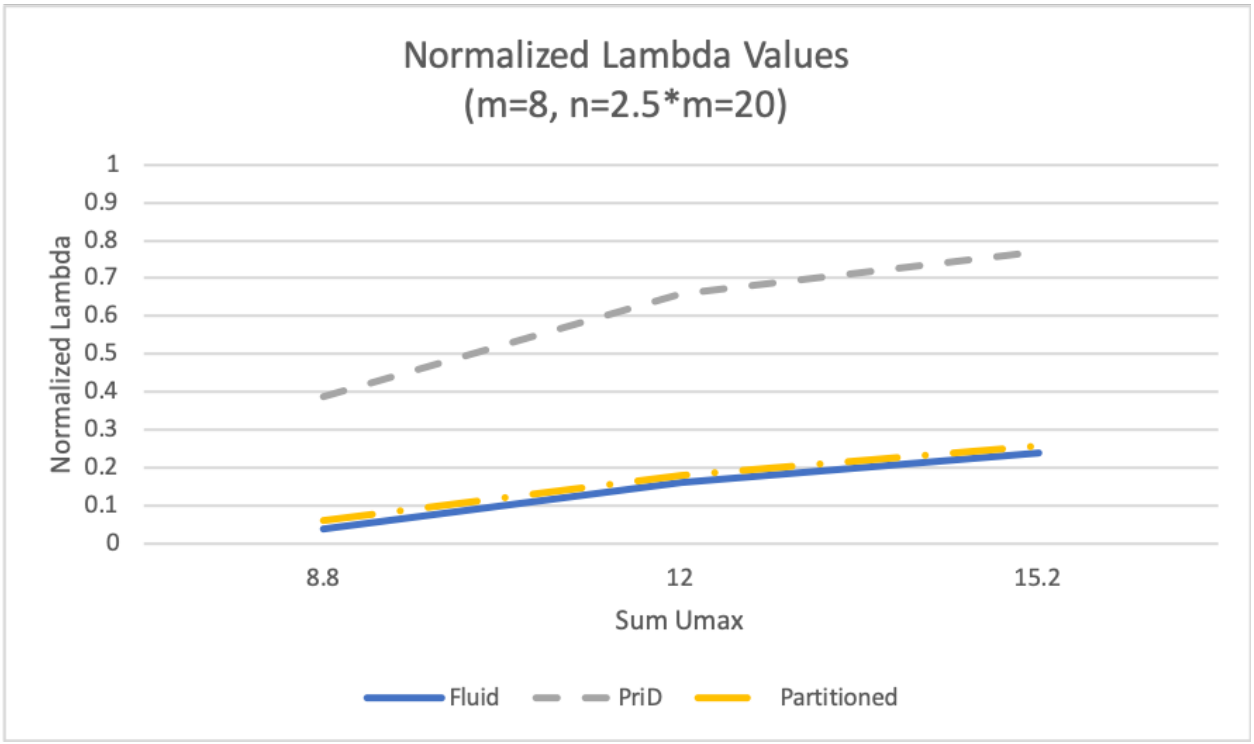


Figure 4.4: Normalized Lambda Values



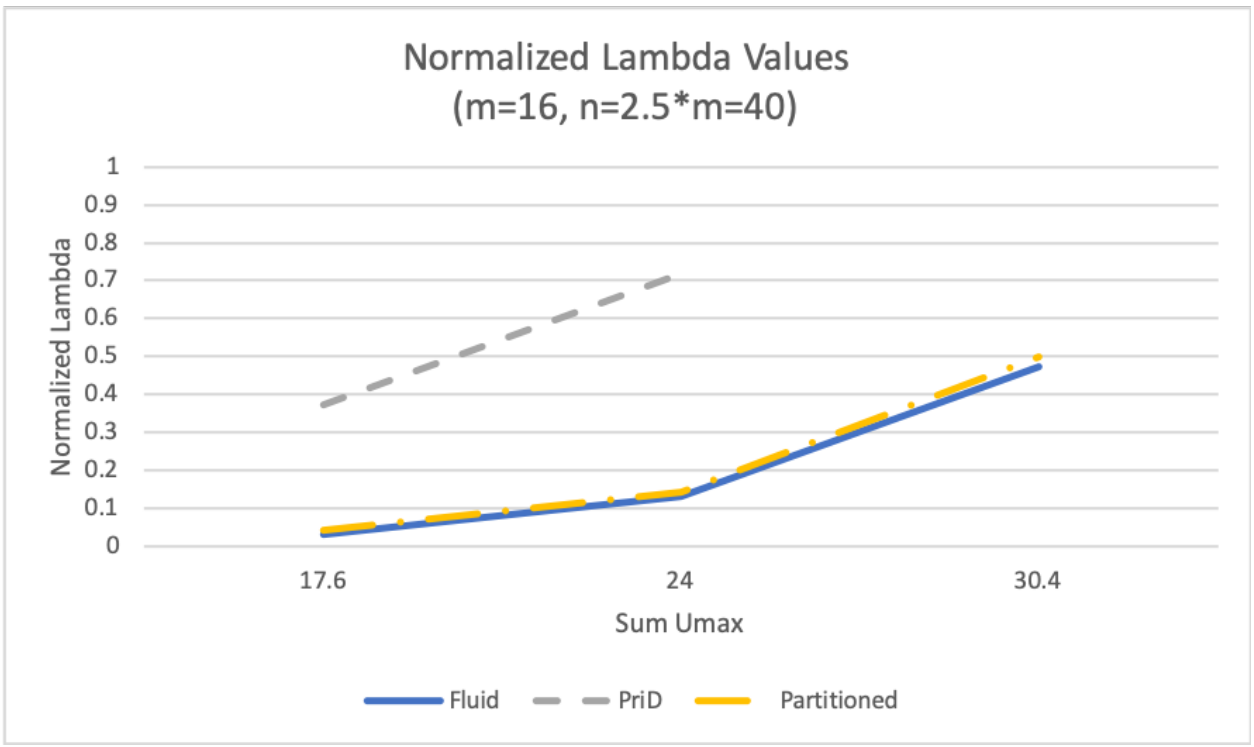
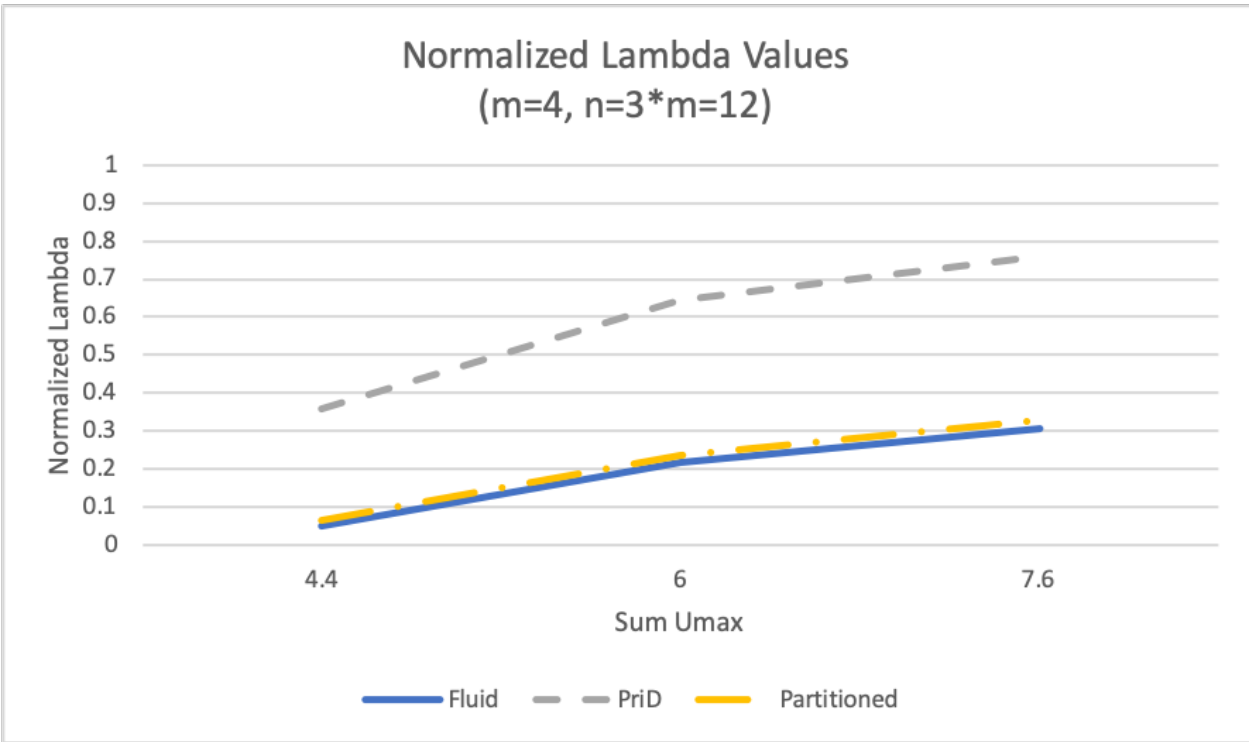


Figure 4.5: Normalized Lambda Values



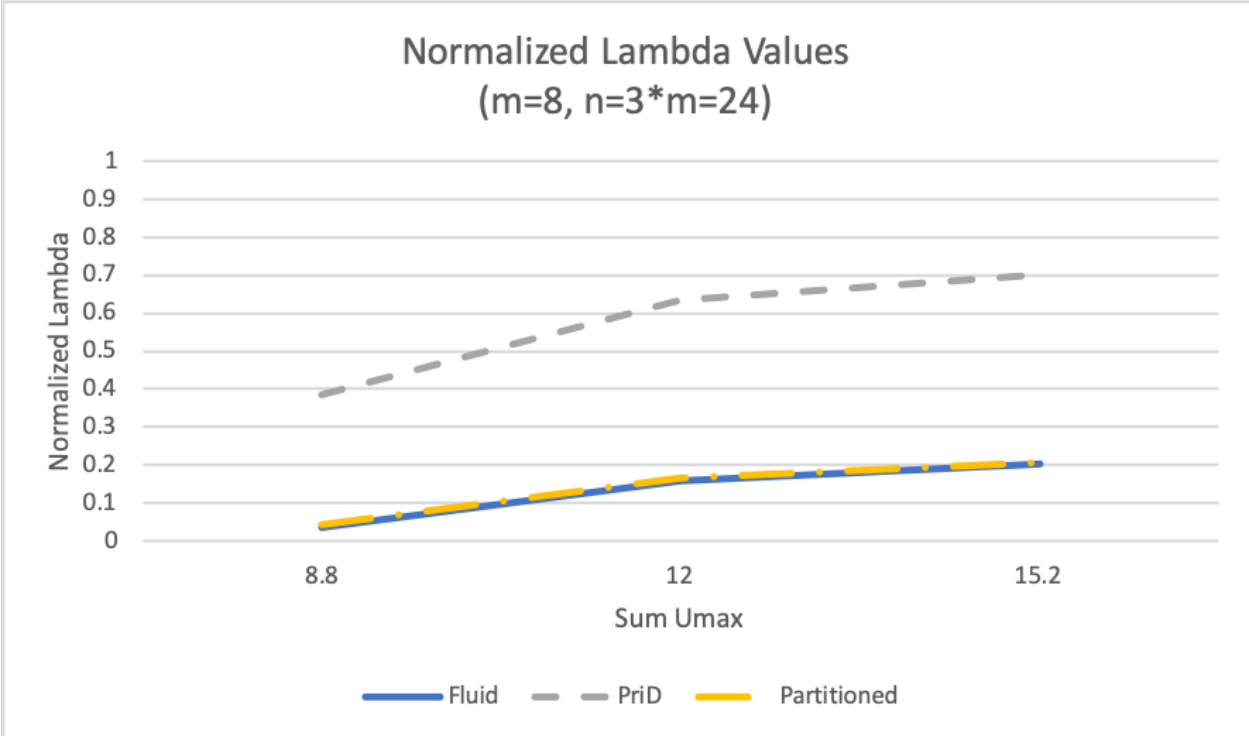
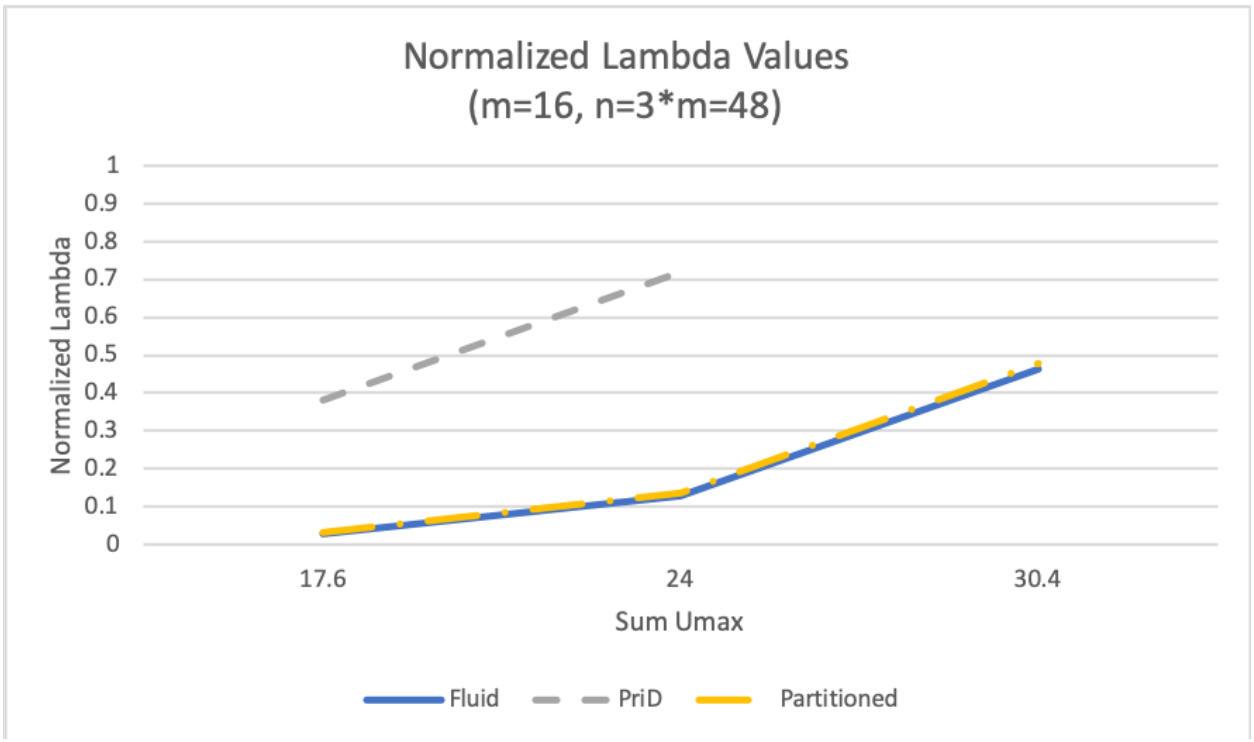


Figure 4.6: Normalized Lambda Values



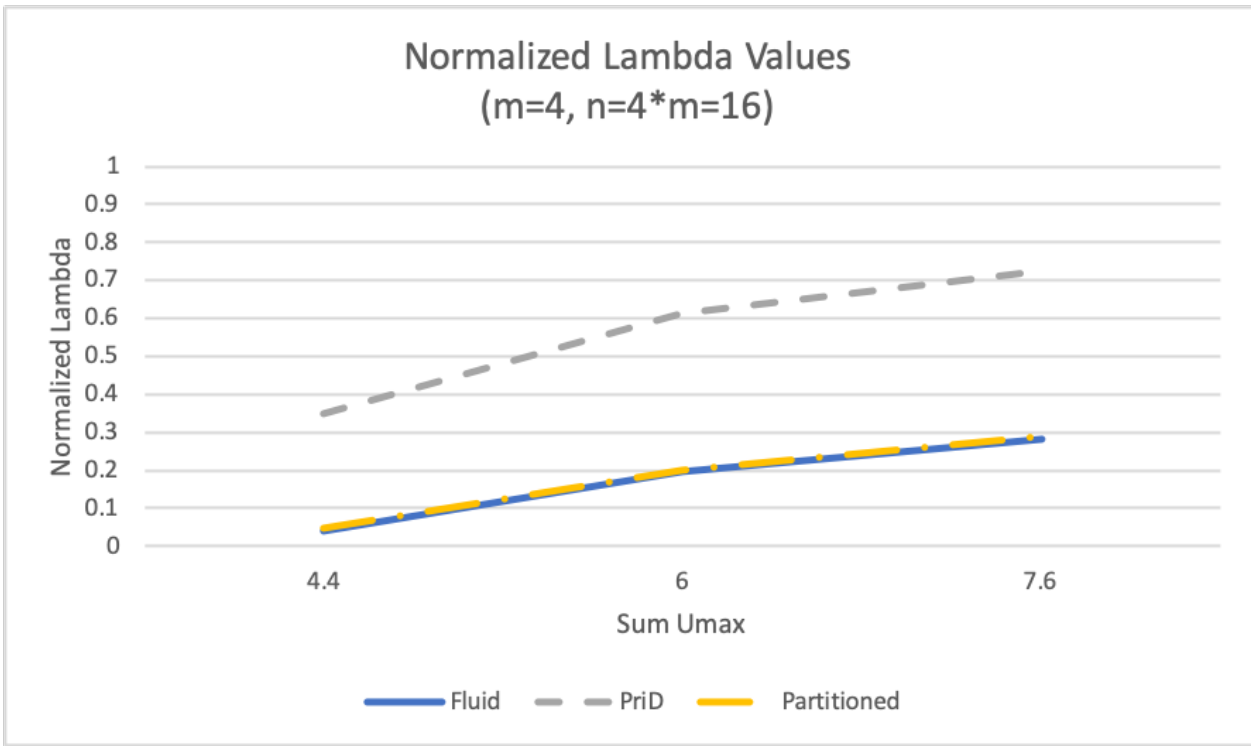
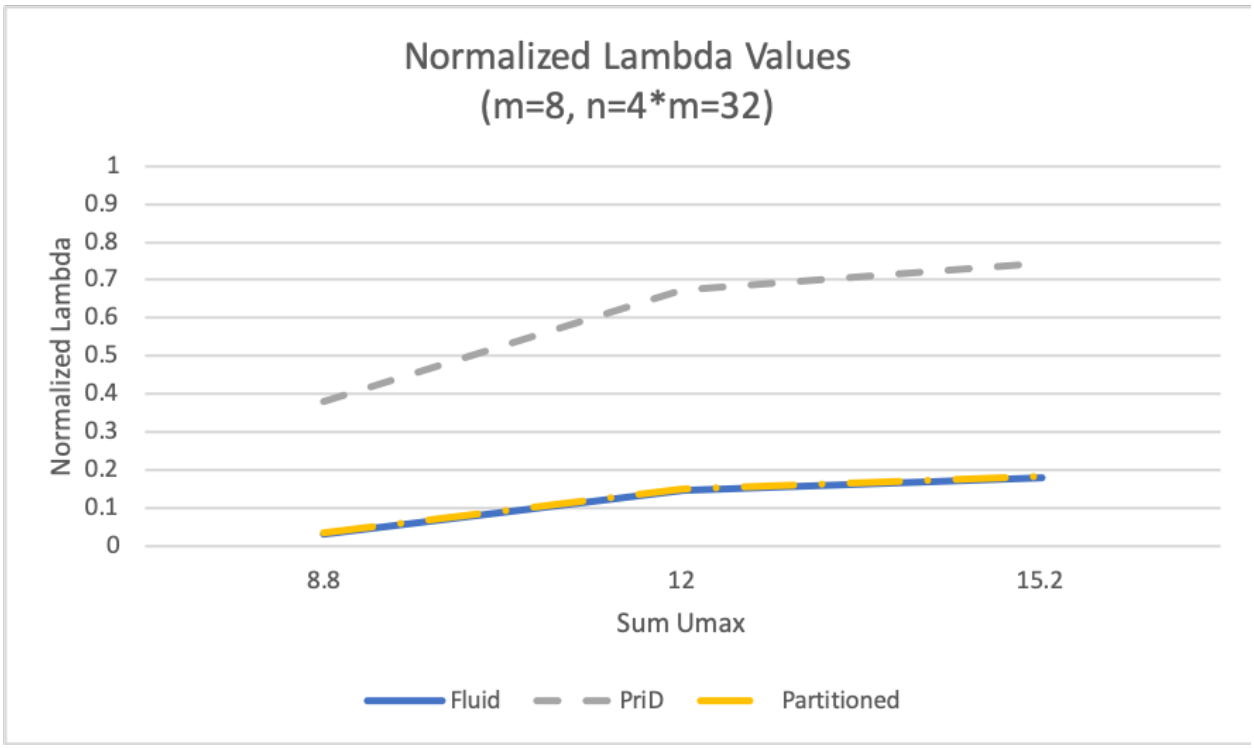


Figure 4.7: Normalized Lambda Values



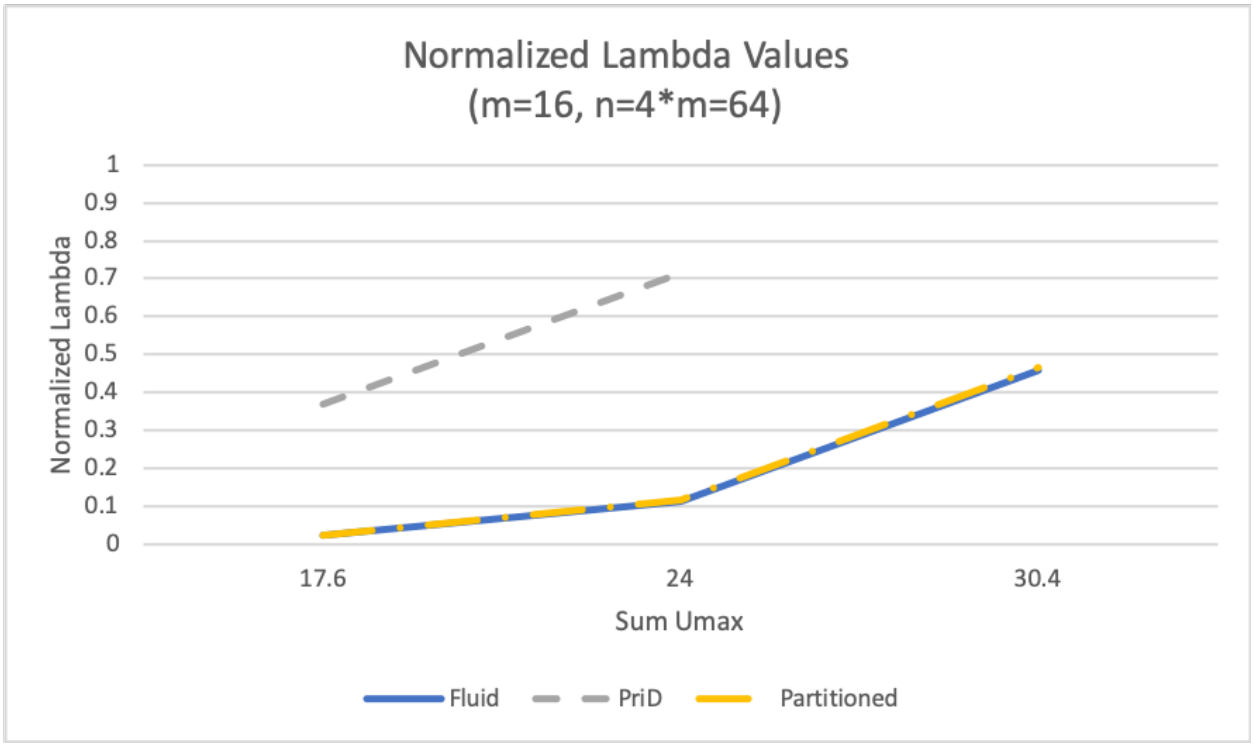
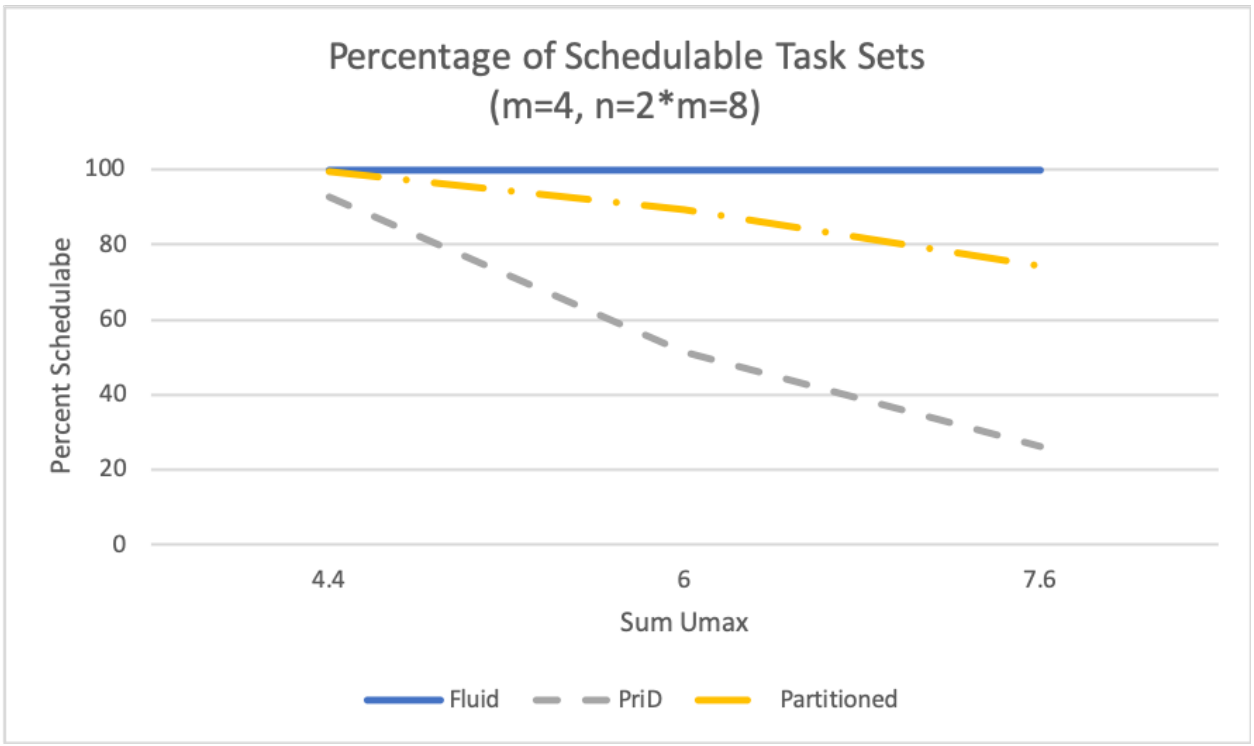


Figure 4.8: Normalized Lambda Values



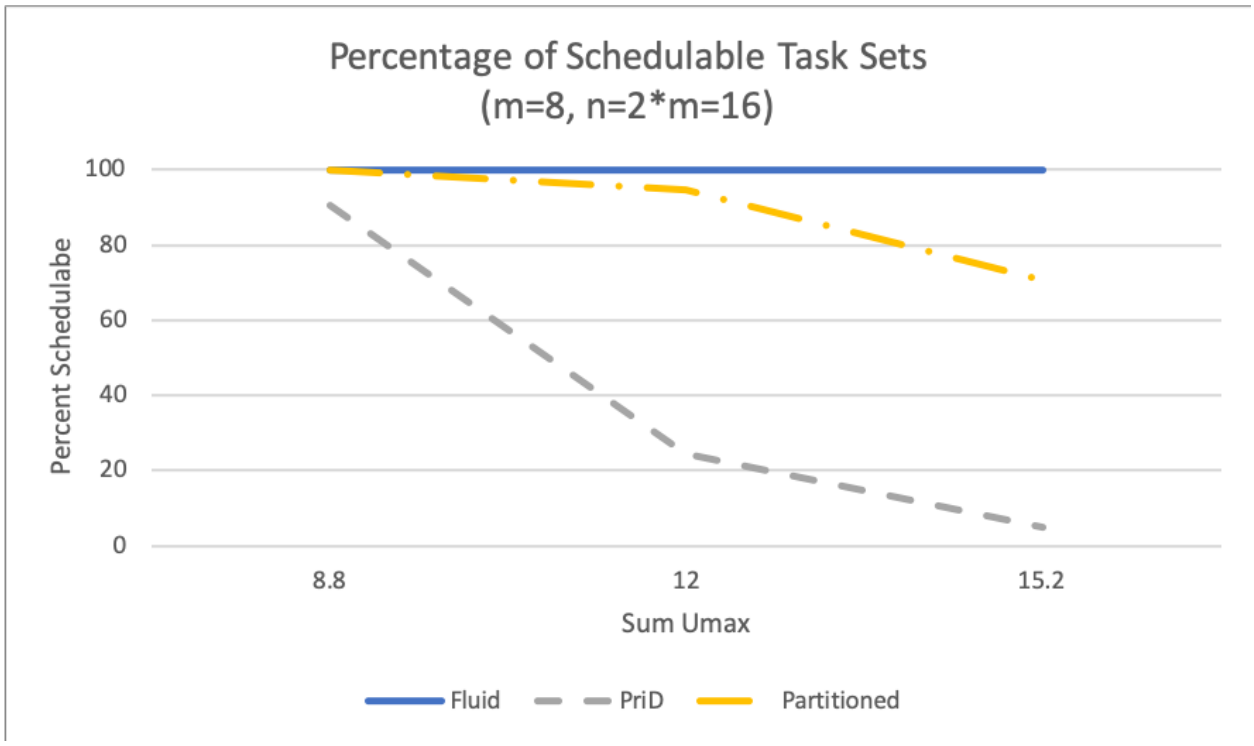
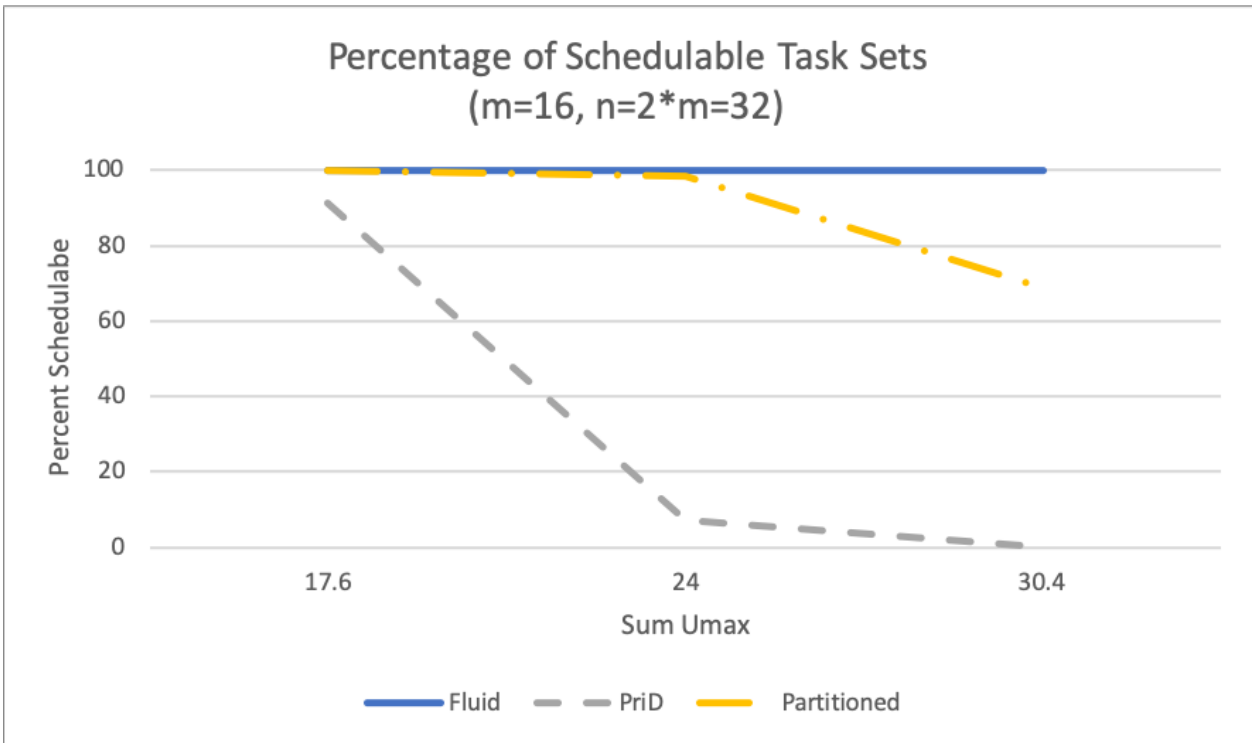


Figure 4.9: Percentage of Schedulable Task Sets



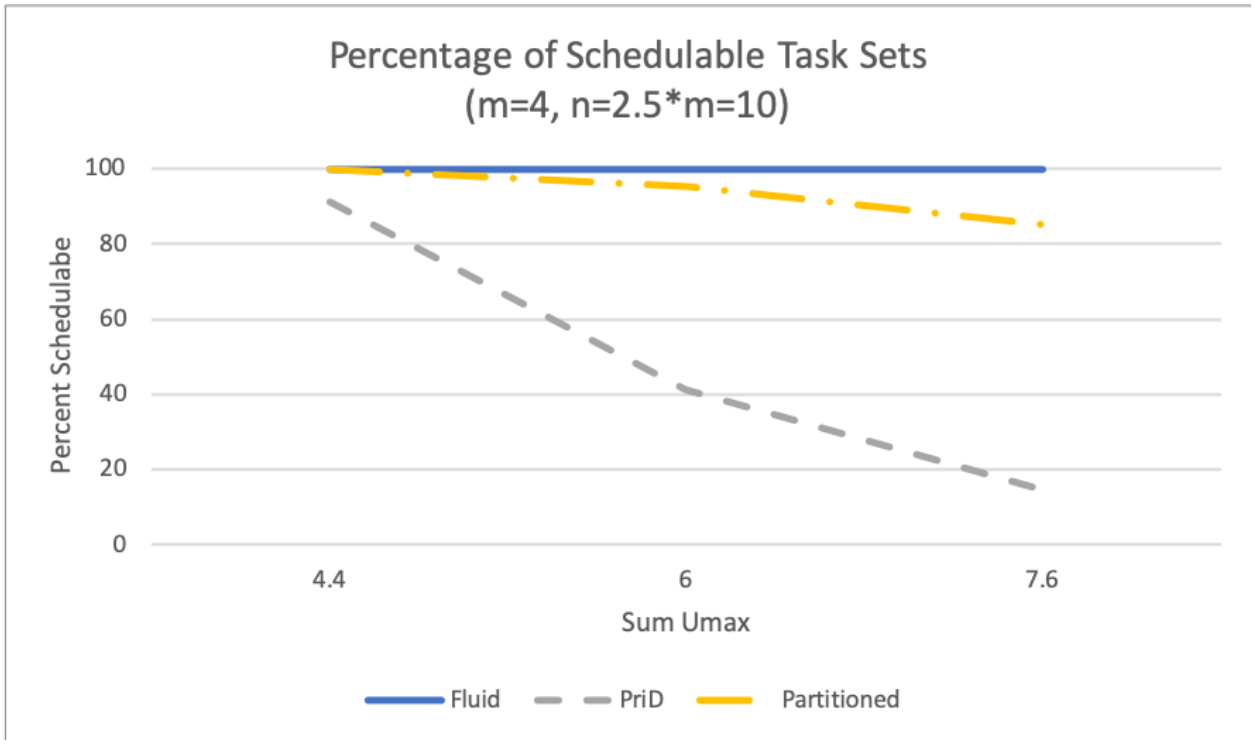
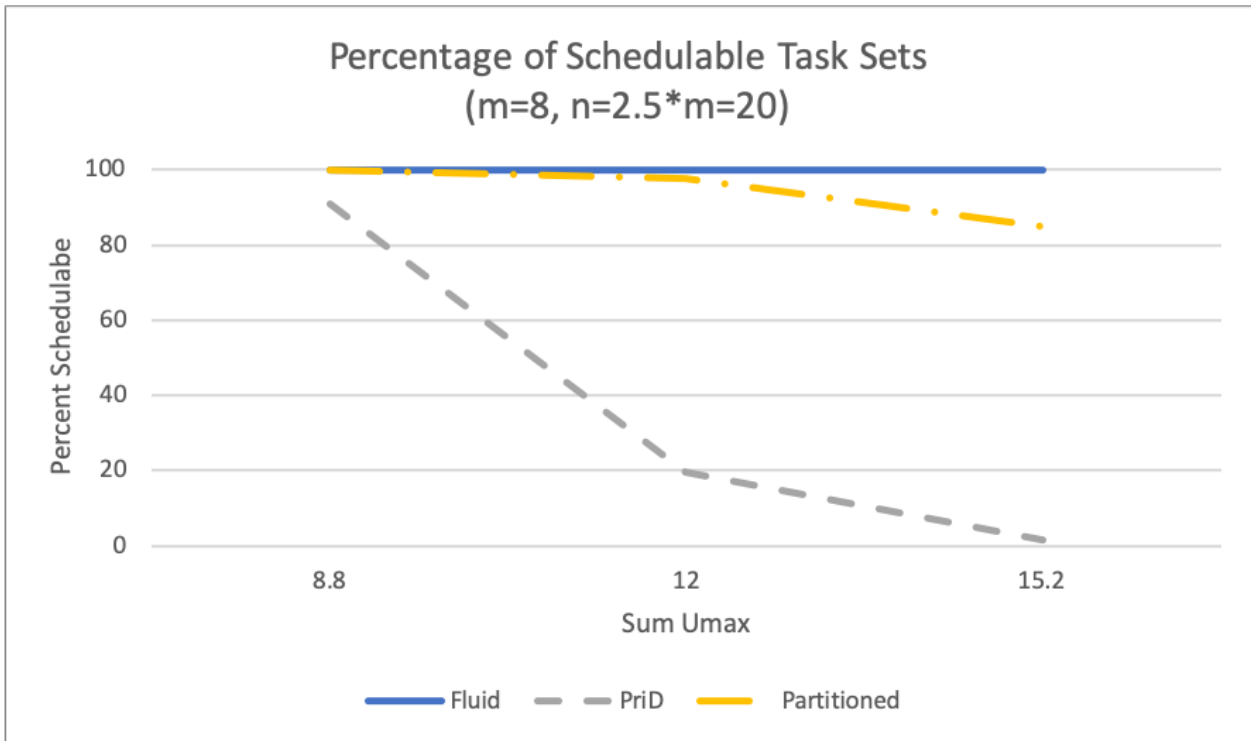


Figure 4.10: Percentage of Schedulable Task Sets



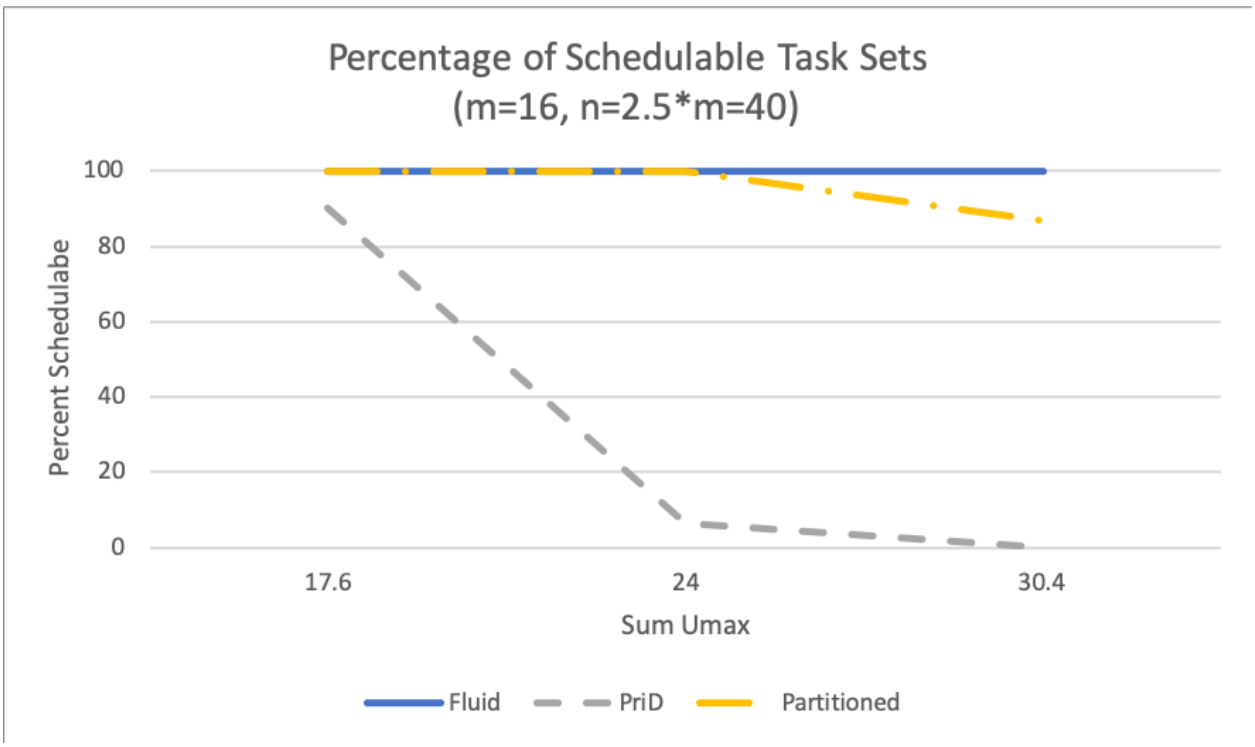


Figure 4.11: Percentage of Schedulable Task Sets

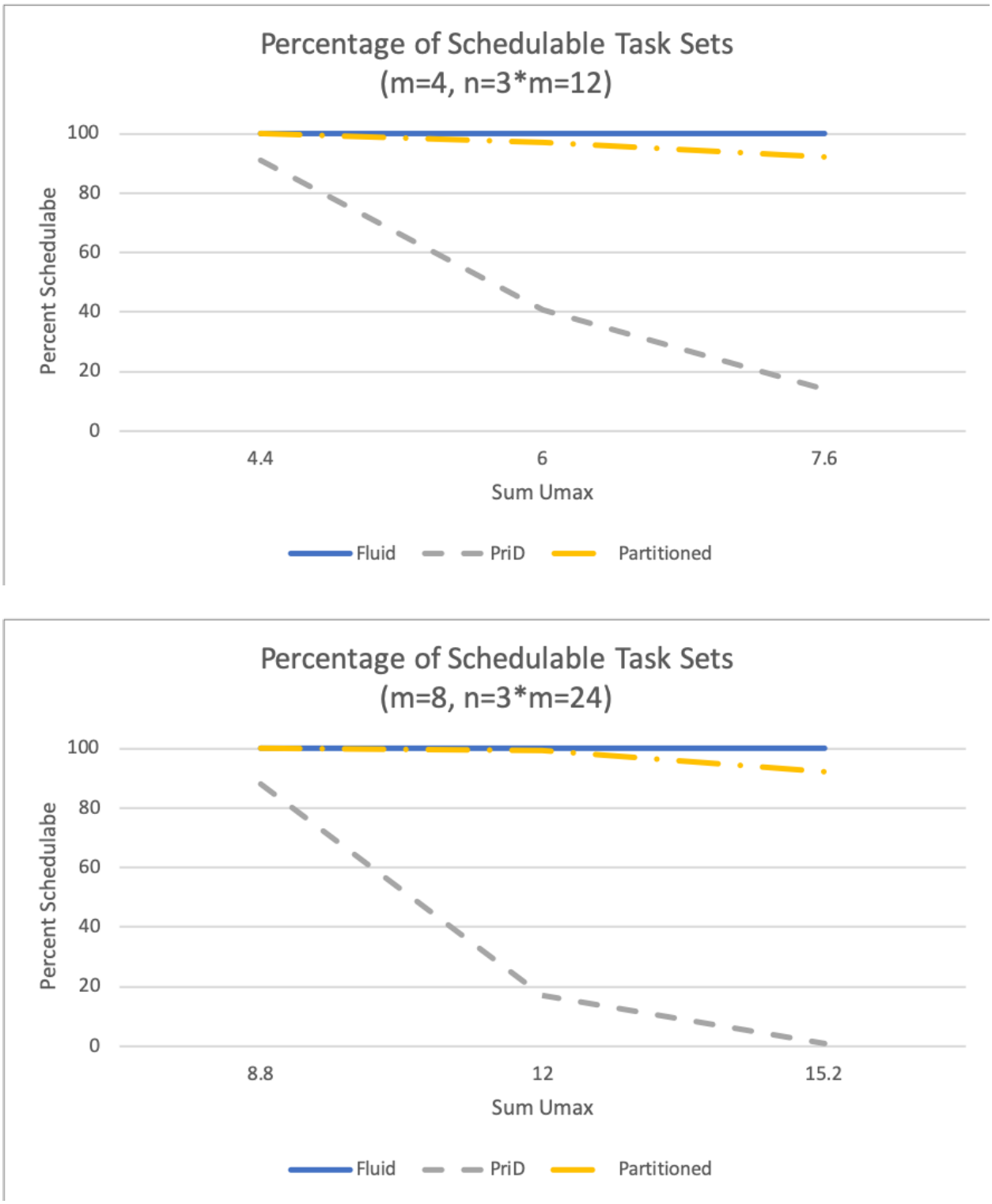


Figure 4.12: Percentage of Schedulable Task Sets

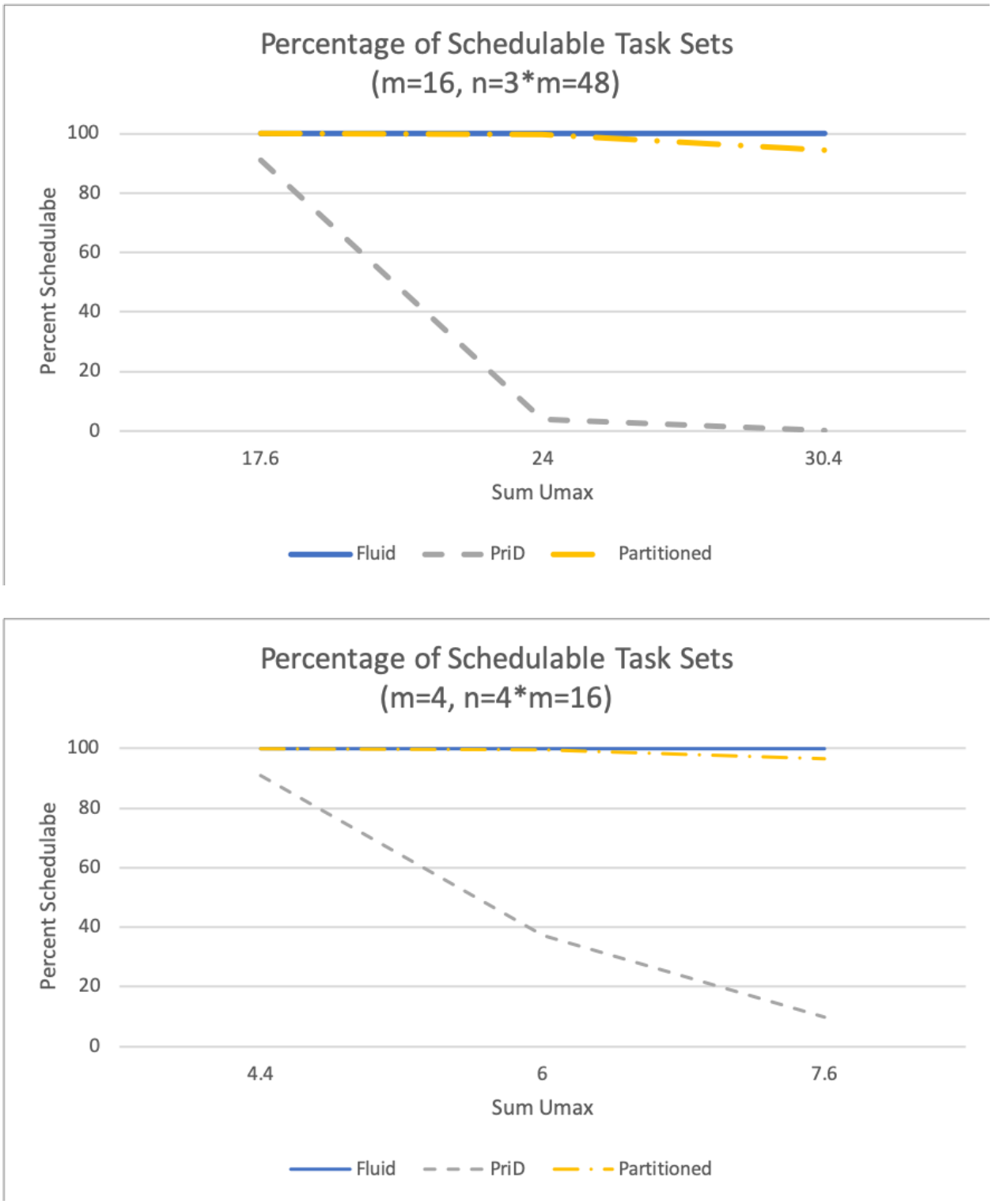
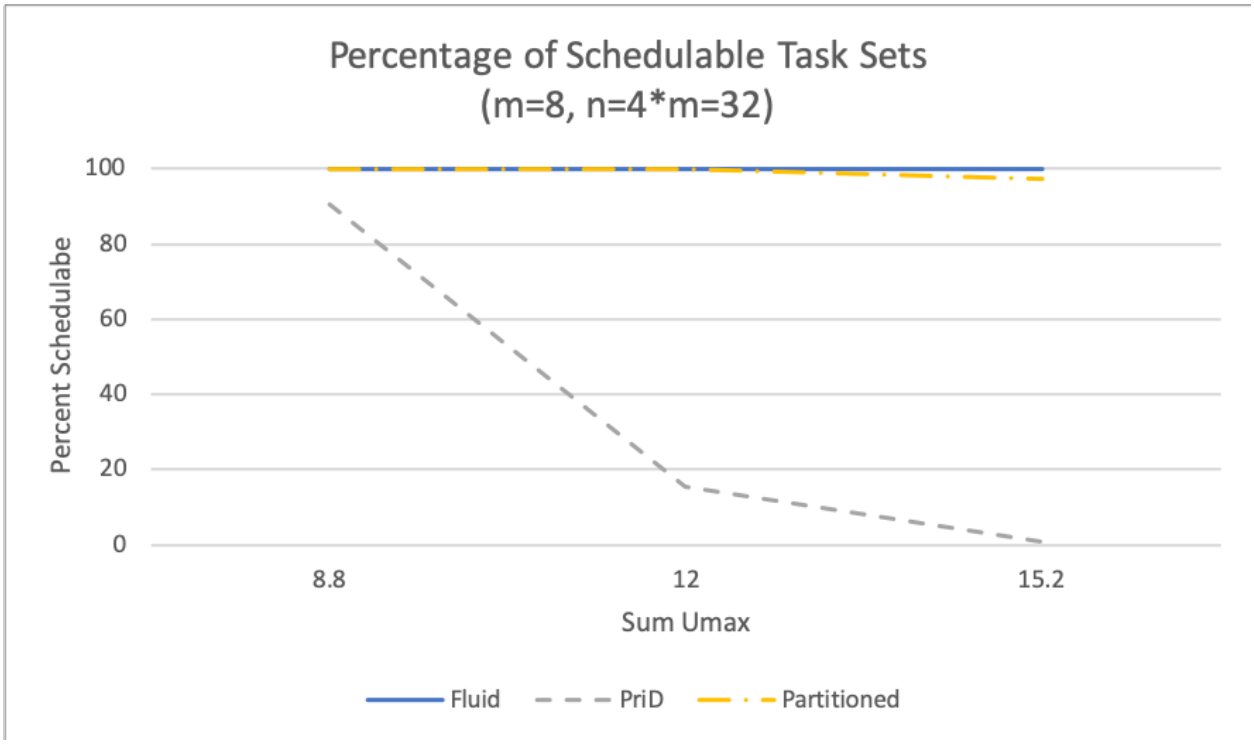


Figure 4.13: Percentage of Schedulable Task Sets



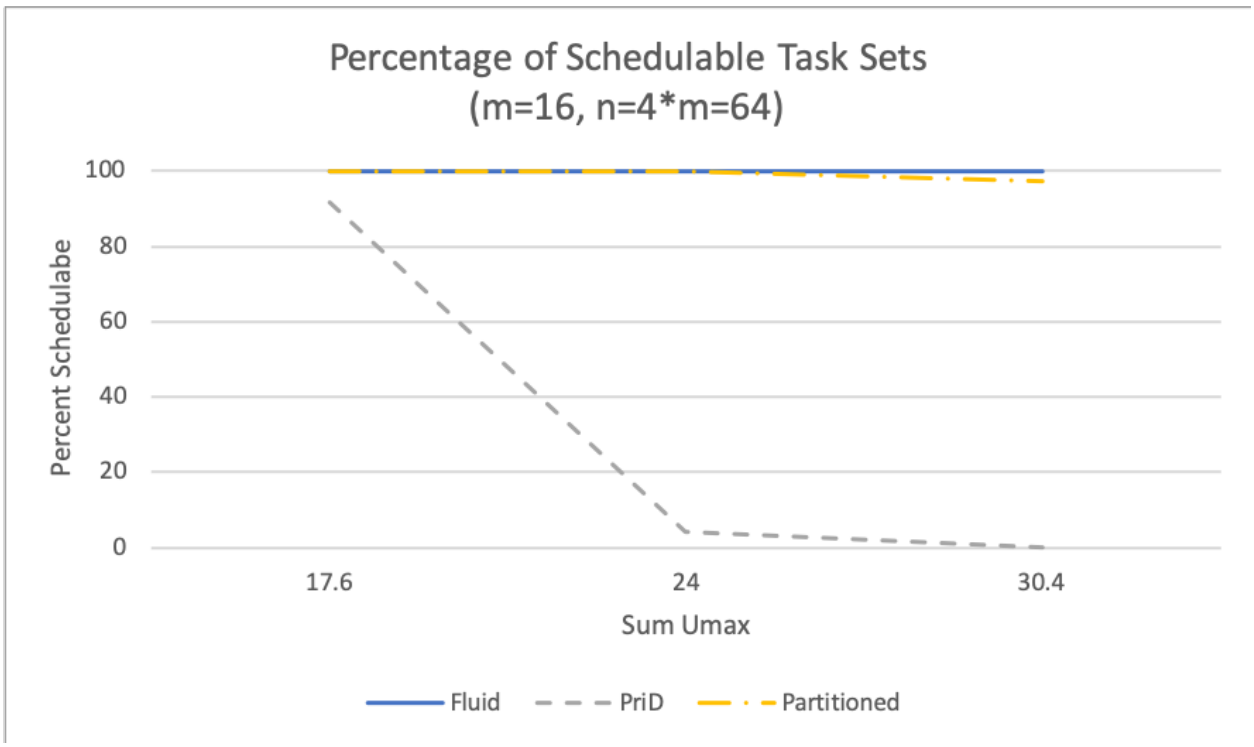


Figure 4.14: Percentage of Schedulable Task Sets

4.6 Summary

In this chapter we have introduced elastic scheduling for sequential tasks on multiprocessor systems. We have introduced algorithms for scheduling such tasks under both global (in a variety of manners) and partitioned scheduling paradigms. We ran an extensive simulation to compare these methods and conclude that partitioned scheduling should be used if possible.

Chapter 5

Computational Elasticity

In this chapter we further expand the elastic model to allow tasks to adapt their computational workloads in place of adapting their periods. We refer to tasks that do so as ***computationally-elastic tasks***, and tasks that adapt their periods as ***period-elastic tasks***. In this chapter we use real-time scheduling of parallel elastic tasks (from Chapter 3) to demonstrate computational elasticity, but the concept is equally applicable to sequential elastic tasks, whether scheduled on a single processor as in earlier work, or on a multi-core system as in Chapter 4 of this dissertation. This section also introduces a run time system for elastic parallel real-time systems that is able to handle both computationally-elastic and period-elastic tasks.

5.1 Introduction

In the original elastic task model for sequential tasks [7], schedulability of a task set is determined by system utilization (i.e., the sum of each task's computational workload divided by its period). Buttazzo et al. manipulate a task's ***period elasticity*** to change tasks'

utilizations as needed to maintain system schedulability, while allowing code e.g., a control algorithm, to perform better when run at a higher periodic rate, or a multi-media player to offer a better picture when run at a higher frame rate.

However, a task can also change its utilization by adapting its computational load instead of its period, i.e., manipulating its ***computational elasticity***. Either an increase in a task's computational load or a proportional decrease in its period will result in the same increase in its CPU utilization. Similarly, CPU utilization can be decreased by decreasing a task's computational load, or increasing its period. Although an increase or decrease in computational load may not make sense for some tasks, other tasks (e.g., a simulation that must iterate at a constant rate but whose accuracy depends on how much it can compute during each iteration) can use this capability to adjust their quality of service.

Although exploiting computational elasticity is possible in sequential tasks, it is perhaps more relevant for parallel tasks, where computational workload can be increased (while maintaining a constant period) by providing additional CPUs for the task to utilize (which sequential-task scheduling cannot do).

Towards a more comprehensive treatment of elasticity in parallel real-time tasks, this chapter introduces and discusses the novel concept of computational elasticity in parallel real-time systems. It also identifies and encapsulates an equivalence between period elasticity and computational elasticity by adapting an algorithm from Chapter 3 (originally used to optimally schedule only period-elastic tasks) to now optimally schedule both period-elastic and computationally-elastic parallel real-time tasks.

In this chapter, we also introduce platform support for interchangeable adaptation of either the period or the workload of each parallel real-time task, which allows some tasks to adjust their periods and others to adjust their workloads. We have designed and implemented an

efficient runtime system that, when tasks adaptively change their computational demands or their periods, makes the necessary elastic scheduling changes to other tasks in the system to ensure that all tasks in the task set remain schedulable. We also demonstrate the equivalence of computational and period elasticity by scheduling two task sets in which all tasks are identical, with the exception of one task in each task set: in one task set this designated task modifies its period while in the other it modifies its computational load, within the same minimum and maximum utilization. Results of our evaluation show that the system adapts in the same way, and finds the same optimal schedule regardless of whether the designated task adapts its period or its computational work load.

The rest of this chapter is structured as follows. Section 5.3 introduces the expanded parallel elastic real-time task model and proves that the optimal scheduling algorithm introduced in Chapter 3 for period adaptation only, is easily extended to work correctly whether tasks change their computational workloads or their periodic rates. Section 5.4 presents the design of our elastic parallel real-time runtime system and Section 5.5 evaluates its effectiveness. Section 5.6 summarizes the chapter.

5.2 Background and Related Work

In this chapter we expand our earlier work on elastic parallel-real time scheduling to support adaptive computational workloads. This section describes single-core elastic scheduling and the federated scheduling paradigm, and gives an overview of our prior work with ***period-elastic*** parallel real-time tasks.

5.2.1 Elastic Scheduling

The elastic task model itself is a generalization of the sporadic task model. Each of n sporadic tasks $\tau_i = (C_i, T_i^{(min)}, T_i^{(max)}, E_i)$ has a **current** *minimum inter-arrival time* (or *period*) T_i and a constant workload represented by its worst-case execution time (WCET) C_i. Under elastic scheduling each task's period T_i can vary over the range of $[T_i^{(min)}, T_i^{(max)}]$ where $T_i^{(min)}$ is the preferred period and $T_i^{(max)}$ is the slowest acceptable period. To ensure system-wide schedulability, overall system *utilization* $\sum_{i=0}^{n} \frac{C_i}{T_i}$ must remain below a desired utilization U_d (e.g., 1.0 for single-core preemptive EDF scheduling). Each task's period is lengthened from $T_i^{(min)}$ (thereby reducing task utilization) proportionally to its current utilization and *elastic coefficient* E_i. The elastic coefficient is again a measure of a task's ability to change its period, similar to a spring's ability to be expanded or contracted. The higher the value of E_i, the more elastic a task, and the more able it is to change its period.

In the original elastic scheduling work [7] Buttazzo et al. presented an efficient ($\Theta(n^2)$) iterative algorithm for task period selection when the system needed to adapt, which (if possible) finds each task τ_i an appropriate period T_i such that $\sum_i (C_i/T_i) \leq U_d$ and $T_i^{(min)} \leq T_i \leq T_i^{(max)}$ for all tasks τ_i. (As stated above U_d is a threshold defined according to the scheduling algorithm that is used; for single-core preemptive EDF scheduling $U_d = 1.0$.) Chantem et al. [12, 13] proved that the iterative algorithm from [7] is exactly equivalent to solving the following optimization:

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

such that:

$$U_i^{(min)} \leq U_i \leq U_i^{(max)} \quad \text{for all } \tau_i, \text{ and}$$

$$\sum_{i=1}^{n} U_i \leq U_d$$

where $U_i^{(max)} = \frac{C_i}{T_i^{(min)}}$ represents the maximum possible utilization of a task obtained from running at period $T_i = T_i^{(min)}$.

The original work involving elastic tasks [7] assumed *implicit deadlines* in which $D_i = T_i$, but theory involving that model has since been expanded to include: *constrained deadlines* in which $D_i \leq T_i$ [13], resource sharing [8], and unknown computational load [10]. Our work in this chapter explores a similar (but orthogonal) direction to that in [10] except that we assume a variable, yet known and controlled workload.

5.2.2 Federated Scheduling

The *federated scheduling* paradigm is used to schedule sporadic parallel real-time tasks on multiple processors. It was designed by Li et al. [36] to schedule tasks whose computational demand is such that a single processor cannot possibly guarantee schedulability. As such, a set of processors is assigned to be used exclusively by each parallel real-time task whose computational requirements exceed the capacity of a single CPU.

Each task τ_i again has a *minimum inter-arrival time (or period)* T_i and (in this chapter) an *implicit deadline* $D_i = T_i$. The computational requirements of each task are represented as a *directed acyclic graph (DAG)* in which each node is a block of sequential computation, and no node can be performed until all of its predecessors have finished execution. Any two nodes whose predecessors have all finished running can be executed in parallel. In place of the WCET parameter used in sequential tasks (such as for Buttazzo's elastic scheduling model), federated scheduling represents the computational workload of a DAG task with total *work* C_i, and *critical path length (or span)* L_i. The work represents the sum of the workloads of

each of node in the DAG task. In other words, it is the amount of time it would take the DAG to run sequentially on a single processor. Similarly, the span is the highest-weighted (by summation of computation time) chain of nodes in a DAG. Because the nodes making up such a chain must be run sequentially, the span of a DAG represents the amount of time that the task would need to run on a theoretically infinite number of processors.

In federated scheduling, tasks that require more than one processor (i.e., utilization $U_i > 1.0$) are referred to as *high utilization* tasks. Similarly, tasks that can be scheduled feasibly on a single processor ($U_i \leq 1.0$) are *low utilization* tasks.

5.2.3 Parallel Real-time Elastic Scheduling

Our work in Chapter 3 has expanded Buttazzo's elastic model to include parallel tasks scheduled under federated scheduling. Each task τ_i is characterized by the parameters

$$\tau_i = (C_i, L_i, U_i^{(max)}, U_i^{(min)}, E_i) \tag{5.2}$$

where $U_i^{(max)} = C_i/T_i^{(min)}$ and $U_i^{(min)} = C_i/T_i^{(max)}$ are the maximum (i.e., desired) and minimum acceptable utilization values, respectively. Note that using minimum and maximum utilization values is functionally equivalent to characterizing the task by a maximum and minimum period, but we use the utilization parameter as it is more directly applicable to our scheduling algorithm. As before, C_i and L_i are the *work* and *span* of a parallel task represented by a DAG, and E_i is the elastic coefficient of a task, representing the ease with which a task's period can be changed, analogous to a spring's resistance to being compressed. Note that in the scheduling of sequential processes (including scheduling of traditional elastic tasks), system-wide utilization (and by extension individual tasks' utilization) is used directly to determine whether a task set is schedulable. However, under federated scheduling, a task

set is considered schedulable if and only if the m processors available to the system are enough to assign each task its requisite number of dedicated processors, (largely) independent of the utilization of individual tasks. It was proved in [23] that the makespan of the schedule for a given DAG is guaranteed to be no larger than the difference between the work and span divided by the number of processors available, plus the span:

$$\frac{C_i - L_i}{m} + L_i \tag{5.3}$$

Therefore an upper bound on the makespan for a DAG may be stated in terms of only its work and span parameters. Equivalently, if the DAG represents a real-time piece of code characterized by a relative deadline parameter D, then $(\frac{C_i - L_i}{m} + L_i) \leq D$ is a sufficient test for determining whether the code will complete by its deadline upon an m-processor platform. Because we assume implicit deadlines with $D_i = T_i$, we can show

$$\begin{aligned} & \frac{C_i - L_i}{m_i} + L_i \leq T_i \\ \Leftrightarrow \quad & \frac{C_i - L_i}{m_i} \leq T_i - L_i \\ \Leftrightarrow \quad & m_i \geq \frac{C_i - L_i}{T_i - L_i} \end{aligned}$$

Under federated scheduling, since the number of processors assigned each task is an integer, we therefore have

$$m_i = \left\lceil \frac{C_i - L_i}{T_i - L_i} \right\rceil \tag{5.4}$$

Therefore a task set is schedulable under federated scheduling if and only if $\sum_{i=1}^{n} m_i \leq m$.

As such our work in Chapter 3 solves the parallel version of the optimization equation given in Definition (5.1), namely:

$$\textbf{minimize} \sum_{i=1}^{n} \frac{1}{E_i}(U_i^{(max)} - U_i)^2 \tag{5.5}$$

such that:

$$U_i^{(min)} \leq U_i \leq U_i^{(max)} \text{ for all } \tau_i \text{ and}$$

$$\sum_{i=1}^{n} m_i \leq m$$

In Chapter 3 we also presented a greedy algorithm that optimally solves Definition (5.5) where the period may be varied. In this chapter we modify that algorithm to schedule tasks with elastic computational loads ***or*** periods: Algorithm 5 in Section 5.3.

5.3 Computational Elasticity

This section further develops the concept of task elasticity, originally presented as sequential real-time tasks' ability to dynamically adapt their periods by Buttazzo et al. [7] and expanded work Chapter 3 to allow similar period-only adaptation in parallel real-time tasks. We refer to such tasks as ***period-elastic tasks***. This section further extends those prior results to allow for tasks to adapt their computational load instead of period. We refer to these as ***computationally-elastic tasks***.

In this section we demonstrate that both period-elastic and computationally-elastic tasks can be encapsulated under *elasticity of task utilization* and thereby form an equivalence relationship. Utilization-based scheduling algorithms used in prior work [7, 8, 12, 13] to

schedule period-elastic tasks then can be modified in order to also schedule computationally-elastic tasks.

5.3.1 Computationally-Elastic Task Model

Our computationally-elastic task model in this chapter is similar to the period-elastic task model from Chapter 3 shown in Definition 5.2. However, because we now consider tasks in which adaptation is driven by computational elasticity instead of period elasticity, we replace the fixed computational load C_i with a fixed period T_i and again allow utilization to vary between maximum (desired) and minimum values, which in turn implies a variable workload C_i. This gives our new computationally elastic task model

$$\tau_i = (T_i, L_i, U_i^{(max)}, U_i^{(min)}, E_i) \tag{5.6}$$

in which the *minimum inter-arrival time (or period)* T_i is fixed. We again assume an *implicit relative deadline* $D_i = T_i$, and each task is again represented by a directed acyclic graph (DAG) with (fixed) *critical path length* (or *span*) L_i and overall *work* C_i. However, instead of a constant C_i, each task now has a range of acceptable work values $[C_i^{(min)}, C_i^{(max)}]$ similar to the range of periods found in period-elastic tasks. This range of acceptable C_i values is encapsulated in Definition (5.6) by the interval of acceptable task utilizations $[U_i^{(min)}, U_i^{(max)}]$ where

$$U_i^{(min)} = \frac{C_i^{(min)}}{T_i}$$

and

$$U_i^{(max)} = \frac{C_i^{(max)}}{T_i}.$$

In computationally-elastic tasks, *elastic coefficient* E_i indicates a task's ability to have its computational load changed. A higher value of E_i indicates a task whose C_i value is more easily changed.

The directly proportional relationship between the minimum and maximum computational load ($C_i^{(min)}$ and $C_i^{(max)}$) and the minimum and maximum utilization ($U_i^{(min)}$ and $U_i^{(max)}$) of computationally-elastic tasks provides the key insight for adapting existing period-elastic scheduling techniques to also schedule computationally-elastic tasks. A similar (yet inverse) relationship exists between period and utilization of period-elastic tasks. Since both computational-elasticity and period-elasticity can therefore be encapsulated as utilization, it follows that either could be scheduled under a utilization-based scheduling algorithm. This subsection explores the adaptation of existing utilization-based scheduling algorithms used previously for scheduling exclusively period-elastic tasks to now schedule **both** period-elastic **and** computationally-elastic tasks.

As in our prior work with parallel real-time period-elastic tasks Chapter 3, we schedule high-utilization tasks using federated scheduling. Recall that under federated scheduling any task in which $U_i > 1.0$ is considered a ***high-utilization task***, and likewise any task in which $U_i \leq 1.0$ is a ***low-utilization task***. In this chapter we assume that all tasks are always only high-utilization or low-utilization: a task's elastic nature and adaptive period or computational load cannot carry it across the boundary from one category to the other.

5.3.2 Scheduling of Low-Utilization Computationally-Elastic Tasks

Because low-utilization tasks have utilizations less than 1.0, they can be scheduled like sequential tasks. We now show that the algorithms presented by Buttazzo et al. [7, 8] can be adapted to schedule low-utilization computationally-elastic tasks.

The $Task\_Compress(\Gamma, U_d)$ algorithm presented in [7] converts period-elastic tasks in task set Γ to their utilization-based abstraction. It then compresses each task's utilization proportionally to its elastic coefficient (to the extent that each task can be compressed) until the summed utilization is at or below the desired system utilization U_d. Once the desired system utilization has been achieved, the system is guaranteed to be scheduleable. Each task's compressed utilization becomes its assigned utilization, and the task is assigned to run with the corresponding period T_i.

We showed in the previous subsection that computationally-elastic tasks can also be encapsulated as having a minimum and maximum utilization. Therefore, any computationally-elastic task can have its utilization compressed as is done in the $Task\_Compress$ algorithm discussed above. To successfully schedule computationally-elastic tasks, then, one must simply run the $Task\_Compress$ algorithm and convert utilization to an appropriate work value C_i instead of a period value T_i.

5.3.3 Scheduling of High-Utilization Computationally-Elastic Tasks

For high-utilization computationally-elastic tasks, recall that under federated scheduling each task is assigned a set of dedicated processors. Definition (6.1) determines the number of CPUs needed to schedule a high-utilization task under federated scheduling at a given workload C_i and period T_i. Because we are scheduling computationally-elastic tasks, each task with a constant period T_i but variable workload C_i, may therefore use anywhere from $m_i^{(\min)}$ to $m^{(\max)}$ tasks where

$$m^{(\min)} = \frac{C_i^{(min)} - L_i}{T_i - L_i}$$

and

$$m^{(\max)} = \frac{C_i^{(max)} - L_i}{T_i - L_i}.$$

This matters because under federated scheduling of high-utilization tasks, system utilization does not directly determine whether a task set is scheduleable. Rather, a task set is schedulable if and only if the system has enough processors to give each task enough dedicated processors such that each individual task is schedulable.

In the remainder of this subsubsection we discuss adapting algorithm $Task\_Compress\_Par(\Gamma, U_d)$ from Chapter 3 to include the scheduling of high-utilization computationally-elastic tasks. The resulting algorithm appears in this chapter as Algorithm 5.

$Task\_Compress\_Par(\Gamma, U_d)$ is an optimal (see Chapter 3 for proof) and efficient ($\Theta(n * m + m \log n)$) greedy algorithm that directly solves the optimization equation given in Definition (5.5). The algorithm iteratively assigns a processor to the task that lowers $\sum_{i=1}^{n} \frac{1}{E_i} (U_i^{(max)} - U_i)^2$ the most, with the ultimate goal of assigning processors to tasks until all m processors in the system have been assigned, or all tasks have been given their maximum number of processors while minimizing the above-stated objective.

Although each task has a minimum and maximum number of CPUs on which it can run, $m_i^{(\min)}$ and $m_i^{(\max)}$, we note that any system has a finite number of CPUs m on which to schedule tasks, and all other tasks in the task set also have a minimum number of CPUs on which they can run. Therefore, although a task may be theoretically able to run on up to $m_i^{(\max)}$ CPUs, on any given platform it can be assigned at most $m_{practical_i}^{(\max)}$ where

$$m_{practical_i}^{(\max)} = m - \sum_{i=1}^{n} + m_i^{(\min)}. \tag{5.7}$$

Because our prior work in Chapter 3 focuses on exclusively period-elastic tasks, the remainder of the discussion in this section will focus on computationally-elastic tasks with an

Algorithm 5 Task_compress_par2(Γ, m)

for ($\tau_i \in \Gamma$) **do**
  $m_i^{(\min)} = \lceil (C_i^{(min)} - L_i)/T_i^{(max)} - L_i) \rceil$ ▷ The minimum number of processors needed by τ_i
  $m_i^{(\max)} = \lceil (C_i^{(max)} - L_i)/(T_i^{(min)} - L_i) \rceil$ ▷ The maximum number of processors needed by τ_i
  $m_i = m_i^{(\min)}$
  while $m_i \leq m_i^{(\max)}$ **do** ▷ Compute the desired value for τ_i on each number of processors it could be assigned
    if τ_i is period-elastic **then**
      $T_{(i,m_i)} = \frac{C_i - L_i}{m_i} + L_i$ ▷ $T_{(i,m_i)}$ denotes the shortest (best) period for τ_i if τ_i is given m_i processors
    else ▷ τ_i is computationally-elastic
      $C_{(i,m_i)} = m_i(T_i - L_i) + L_i$ ▷ $C_{(i,m_i)}$ denotes the most computation possible for τ_i if τ_i is given m_i processors
    end if
    $m_i = m_i + 1$
  end while
  $m_i = m_i^{(\min)}$ ▷ Assign τ_i the minimum number of processors it needs
  if τ_i is period-elastic task **then**
    $T_i = T_{(i,m_i)}$ ▷ Assign T_i the corresponding shortest period
  else
    $C_i = C_{(i,m_i)}$ ▷ Assign C_i the corresponding maximum workload
  end if
  $m = m - m_i^{(\min)}$ ▷ m keeps count of processors remaining after minimum needs are satisfied
end for
if ($m < 0$) **then** ▷ There weren't enough processors.
  return UNSCHEDULABLE
else if ($m == 0$) **then**
  return *the processor allocation as determined in the m_i values*
end if
▷ THE REMAINDER OF THIS PSEUDOCODE ALLOCATES THE REMAINING PROCESSORS, ONE AT A TIME

Algorithm 5 Task_compress_par2(Γ, m), (continued)

27: **for** ($\tau_i \in \Gamma$) **do** ▷ Allocate next processor to task that will decrease the optimization objective function the most
28: $x_i = \frac{1}{E_i}(U_i^{(max)} - U_i)^2$
29: $m_i += 1$ ▷ Temporarily assign this processor to τ_i
30: **if** τ_i is period-elastic **then**
31: $T_i = T_{(i,m_i)}$ ▷ Determine corresponding period (previously computed in Line 7 above)
32: **else**
33: $C_i = C_{(i,m_i)}$ ▷ Determine corresponding computational load (previously) compute in Line 9 above
34: **end if**
35: $y_i = \frac{1}{E_i}(U_i^{(max)} - U_i)^2$
36: $z_i = x_i - y_i$ ▷ How much τ_i would decrease sum
37: $m_i -= 1$ ▷ Reclaim temporarily-assigned processor
38: **if** τ_i is period-elastic **then**
39: $T_i = T_{(i,m_i)}$
40: **else**
41: $C_i = C_{(i,m_i)}$
42: **end if**
43: **end for**

Algorithm 5 Task_compress_par2(Γ, m), (continued)

Make a max heap of all tasks, with the z_i values as the key
while m > 0 and heap not empty **do** ▷ Each iteration, assign an additional processor
    $\tau_{most} = heap.pop()$ ▷ This is the task that wold see the most benefit
    $m_{most} = m_{most} + 1$ ▷ Permanently assign processor.
    $m = m - 1$
    if τ_i is period-elastic **then**
        $T_{most} = T_{(most,m_{most})}$
    else
        $C_{most} = C_{(most,m_{most})}$
    end if
    if ($m > 0$ and $m_{most} < m_{most_{max}}$) **then** ▷ Able to receive any more processors?
        $x_{most} = \frac{1}{E_{most}}(U_{most}^{(max)} - U_{most})^2$
        $m_{most} += 1$ ▷ Temporarily assign processor.
        if τ_i is period-elastic **then**
            $T_{most} = T_{(most,m_most)}$
        else
            $C_{most} = C_{(most,m_most)}$
        end if
        $y_{most} = \frac{1}{E_{most}}(U_{most}^{(max)} - U_{most})^2$
        $z_{most} = x_{most} - y_{most}$
        Reinsert τ_{most} into heap
        $m_{most} -= 1$ ▷ Reclaim temporarily-assigned processor
        if τ_i is period-elastic **then**
            $T_{most} = T_{(most,m_most)}$
        else
            $C_{most} = C_{(most,m_most)}$
        end if
    end if
end while
return *the processor allocation as determined in the m_i values*

adaptive work parameter C_i. However, Algorithm 5 encapsulates both period elasticity and computational elasticity in terms of task utilization.

The algorithm begins by determining $m^{(\min)}$ and $m^{(\max)}$ for each task (lines 2-3). We assume that for computationally-elastic tasks (with a fixed period T_i) that $T_i = T_i^{(min)} = T_i^{(max)}$, and likewise for period-elastic tasks (with a fixed computational workload C_i) that $C_i = C_i^{(min)} = C_i^{(max)}$. This allows reuse of the same calculations regardless of the task's elastic nature, as the number of required CPUs increases proportionally with a task's work and inversely with its period.

The algorithm then determines the maximum workload (line 9) that each task can accommodate at each integer over the range $[m^{(\min)}, m^{(\max)}]$ and stores them in a lookup table. The value

$$C_{(i,m_i)} = m_i(T_i - L_i) + L_i$$

comes from solving Definition (6.1) for the value C_i.

$$m_i = \left\lceil \frac{C_i - L_i}{T_i - L_i} \right\rceil \Rightarrow \frac{C_i - L_i}{T_i - L_i} \leq m_i < \frac{C_i - L_i}{T_i - L_i} + 1.$$

Solving both inequalities for C_i, we obtain the interval

$$m_i(T_i - L_i) - T_i + 2L_i < C_i \leq m_i(T_i - L_i) + L_i.$$

Since we always want to perform as much computation as possible, we assign the maximum of

$$C_{(i,m_i)} = m_i(T_i - L_i) + L_i.$$

We then assign each task its minimum number of CPUs (line 13) and corresponding computational load (line 17). If not all CPUs have been assigned to a task, the algorithm continues (lines 27-43) by temporarily assigning an additional processor to each task that has fewer than $m_i^{(\max)}$ CPUs currently assigned, to see how much each task will reduce the sum in Definition (5.5).

In line 44 each task is inserted into a max heap that is indexed on how much each task decreases the sum. We can consider tasks independently because the objective function in Definition (5.5) considers only each task's current utilization $U_i = C_i/T_i$ and maximum utilization $U_i^{(max)} = C_i^{(max)}/T_i$, which are constant ($T_i$) or determined only by each task's number of currently assigned processors ($C_i = m_i(T_i - L_i) + L_i$) and thus independent of other tasks.

Lines 45-72 then repeatedly pop τ_{most}, the task that *most* reduces the objective function in Definition (5.5), from the max heap and permanently assigns it a processor. If the task can still receive more processors, its next potential contribution to the objective function in Definition (5.5) is calculated, and it is reinserted into the heap, until all m processors have been assigned to a task and the algorithm returns. Note that Algorithm 5 decides *how many* processors each task gets, not *which* processors, as we discuss in Section 5.4.

5.4 Concurrency Platform Support

This section presents the concurrency platform we have developed to run ***both*** period-elastic ***and*** computationally-elastic high-utilization tasks. Each task in the system must be either computationally-elastic or period-elastic, but this runtime system allows for the same task set to contain both types of elastic tasks.

We schedule tasks under federated scheduling using Algorithm 5 above. Because of the adaptive nature of elastic tasks, this scheduling algorithm must be rerun any time a system-wide adaptation occurs. Rather than taking away processing time from (and possibly affecting the schedulability of) a task, we dedicate one processor core to running Algorithm 5 and rescheduling the task set when a need arises. Section 5.4.1 discusses the scheduler in greater detail.

The system was built in C/C++ atop Linux with OpenMP. It uses Linux shared memory and POSIX RT Signals for inter-task communication and scheduling. Section 5.4.2 describes the concurrency and synchronization techniques used.

5.4.1 Task Scheduler and Scheduling Mechanisms

Because elastic tasks must adapt on-line, off-line calculation of a static schedule, as is usually done with non-elastic fixed-priority scheduling, only provides an initial schedule, and we must also decide when and how to run an on-line scheduling algorithm.

We dedicate a processor (arbitrarily CPU 0) to running Algorithm 5 and make m (the number of CPUs available to the task set for scheduling) one less than the number of physical processors available in the system. The *Task Scheduler* process begins by running Algorithm 5 to find an initial processor allocation for each task. It then assigns consecutive processors to each task in turn as determined by the results of Algorithm 5 (i.e., beginning by assigning CPU 1 to Task τ_1 and then assigning CPU 2 to Task τ_1 if it needs more CPUs or otherwise assigning it to Task τ_2, etc.). Tasks then run on their assigned processors until a scheduling change must occur. Meanwhile the Task Scheduler process repeatedly polls for whether it needs to reschedule the tasks, and does so as necessary.

We spawn $m^{(\max)}_{practical_i}$ threads for each task as given by Definition (5.7). The first m_i threads (as determined by Algorithm 5) are designated *active* threads and are pinned to the m_i CPUs that are assigned to the task. The remaining $m^{(\max)}_{practical_i} - m_i$ threads are deemed *passive* and are pinned to consecutive processors (beginning with the one immediately after the last processor assigned an active thread) and then made to sleep.

When a new schedule must be found, the Task Scheduler first runs Algorithm 5 again to determine how many processors each task must now be assigned. It then looks at which tasks gained CPUs and which tasks lost CPUs when going from the old configuration to the new and determines which processor will go to which new task. Preference is given to any task with a passive thread already sleeping on a CPU currently occupied by an active thread of a task that must lose a processor. If so, the active thread of the task losing a processor can go to sleep; the passive thread of the task gaining a processor can wake up, and no thread migration occurs.

However, it sometimes unavoidable that the task gaining a processor has no passive threads sleeping on a CPU currently occupied by an active thread of a processor that is losing a CPU. In this case, a passive thread from the task gaining a processor migrates to a processor currently occupied by the task losing a processor. The corresponding formerly active thread then goes to sleep and the thread that migrated becomes active on the CPU.

Figure 5.1a shows the initial state of an example task set in which Task 1 has three CPUs, Task 2 has two CPUs, and Task 3 has 5 CPUs. At some point during execution, however, (as shown in Figure 5.1c) Task 1 notifies the Task Scheduler that it must adapt, and the Task Scheduler runs Algorithm 5 to determine how many CPUs each task should have after the transition. It is determined that Task 1 should gain a CPU at the expense of Task 3, as is

shown in Figure 5.1b. Running Algorithm 5 is $\Theta(n * m + m \log n)$, and (re)assigning CPUs to tasks is $\Theta(n^2)$.

5.4.2 Concurrency and Synchronization

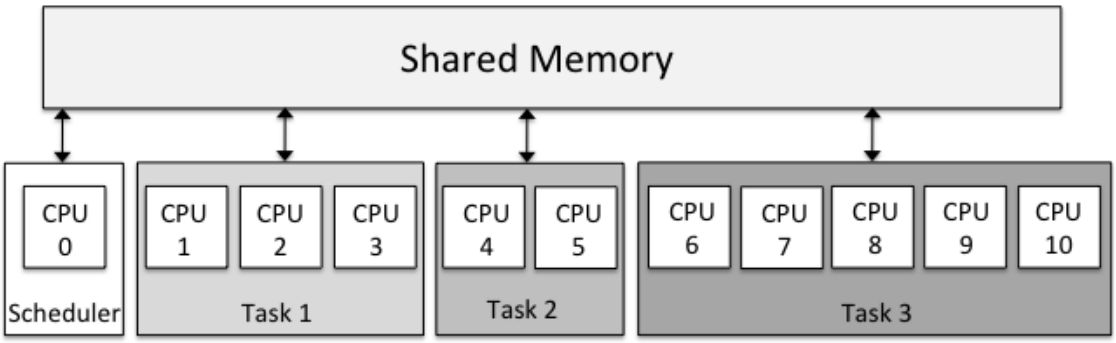
As with any parallel system, concurrency and synchronization are essential. This subsection describes the mechanisms we use to prevent data races and deadlock when each task has not only active threads but also potentially passive threads that are on CPUs currently assigned to other tasks, and transitions in which sole active possession of a CPU is transferred from one task to another.

Shared Memory

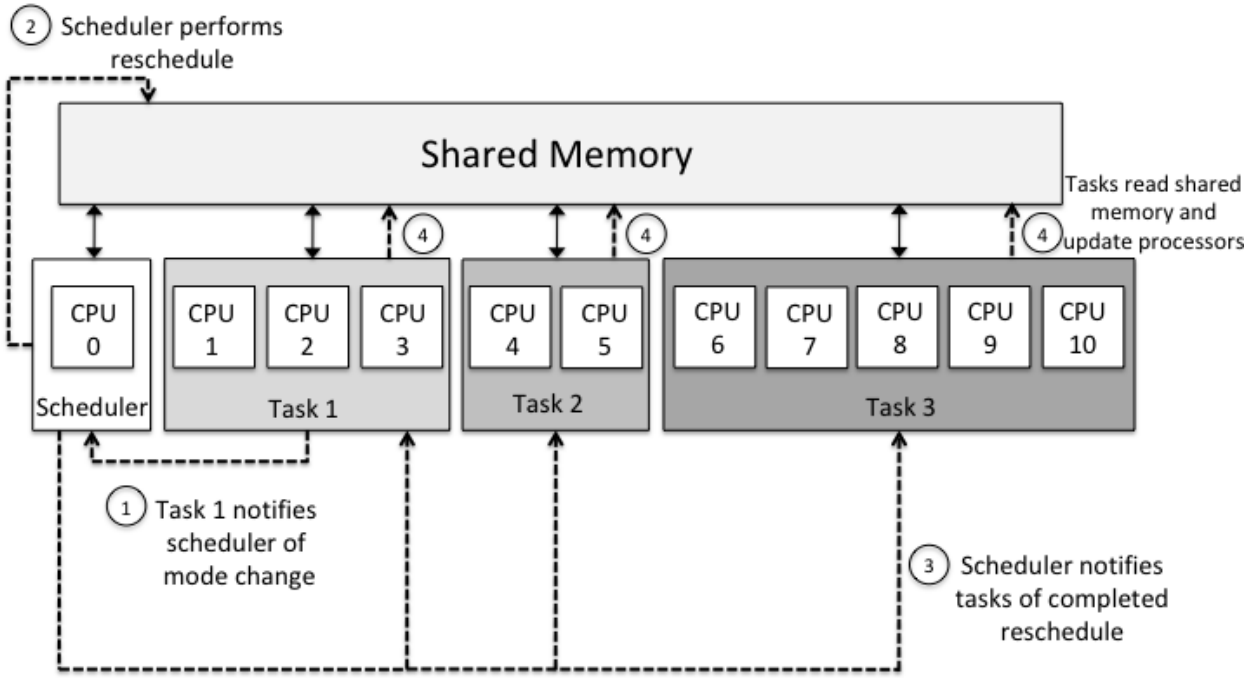
We use Linux shared memory to store all information about task scheduling, including each task's current period, computational workload, and processors with active or passive threads. Each task's data can only be modified by the task itself (e.g., if a task must now run at a certain rate) and by the Task Scheduler, although tasks can read each other's scheduling data if they must. Furthermore, tasks can only write to their own region if the Task Scheduler is currently polling for whether it needs to reschedule, while a re-schedule is occurring. (The task modifying its own data is what ultimately triggers a reschedule in the Task Scheduler.) This therefore ensures that a task and Task Scheduler never attempt to modify the same memory location at the same time.

POSIX RT Signals

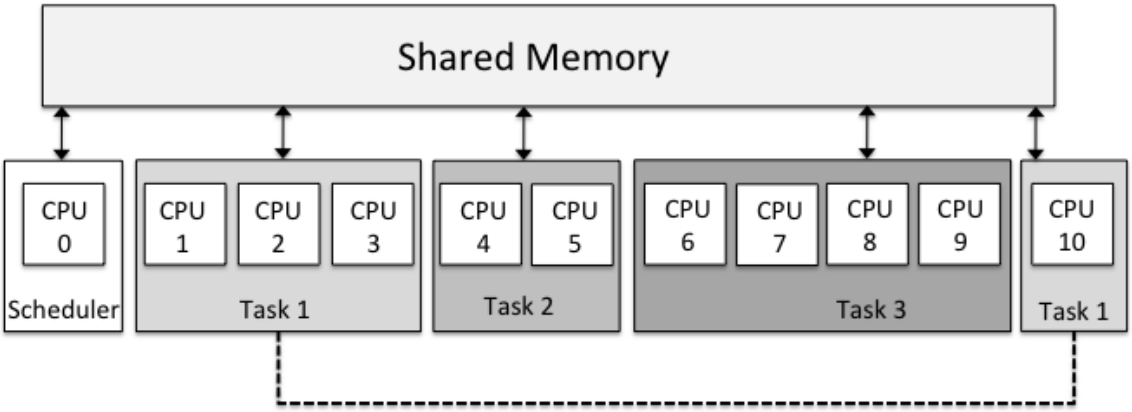
We use POSIX RT Signals for event notification between processes. Unlike standard POSIX signals which are ignored if the task is in the signal handler, RT Signals queue, which



(a) System before reschedule



(b) System during reschedule



(c) System after reschedule

Figure 5.1: Transition of CPUs

guarantees the signal handler is entered each time the signal is received, and no reschedule or adaptation is missed.

Whenever a task has finished modifying its own shared memory region a POSIX RT signal is sent to the Task Scheduler to notify it that a reschedule should occur. Similarly, whenever the Task Scheduler has finished creating the new schedule and assigning processors to tasks, it sends a different POSIX RT signal to each task (in parallel) to notify each task that it needs to read the shared memory region and potentially transition to a new set of CPUs.

Thread Barrier

As mentioned earlier, each task has m_i active threads at any given time, but also $m^{(\max)}_{practical_i} - m_i$ passive threads asleep on other processors. Because we are using OpenMP (OMP) for parallelism, each thread, whether active or passive, must participate in each *#pragma omp parallel* region and is immediately and automatically awoken by OpenMP at the beginning of each iteration. We resolve this issue by implementing a modified version of the parallel barrier introduced previously [35]. The first thing each OMP thread does during a parallel region is reach this barrier. If the thread is designated as active by the task, then the task may proceed past the barrier. However, if the thread is designated as passive, the thread immediately goes back to sleep. The active threads then do the necessary parallel work, and reach the barrier again at the end of the parallel region. The last active thread to reach the barrier then wakes up any sleeping passive threads which race through the already completed parallel work and to the barrier. Just as all threads must enter the parallel region, no thread can leave the *#pragma omp parallel* region until all threads are ready to leave. Therefore, both the barrier and waking up passive tasks are unavoidable. To minimize the amount of time spent in the barrier, each passive thread is given a higher real-time priority than each active thread so that a passive thread can immediately waken, pass through the barrier,

and go back to sleep. The amount of time that each active thread can be interrupted is determined by the number of passive threads sleeping on each task and their task periods. An active thread will be interrupted at the beginning and end of each of the passive task's iterations. This is a very small overhead each time, however ($10 - 20\mu$ sec).

5.4.3 Ensuring a Safe Transition

To ensure no task misses a deadline due to giving up a processor, transitions of CPUs must occur between iterations of a task. At the end of each iteration, a task checks to see whether rescheduling has occurred. If so, it attempts to transition to its new set of CPUs. If the task loses CPUs, it marks all active threads on them as passive and makes those threads sleep. This task can now begin its next iteration with fewer CPUs and its updated period or computational workload, depending on the nature of that task's elasticity. Tasks gaining CPUs, however, cannot take possession of their new CPUs until the prior task has given it up, since under federated scheduling entire CPUs are dedicated to a single task. Therefore, if all of its gained CPUs are not ready, it keeps its prior set of CPUs and begins another iteration under its prior workload or period. The amount of time it takes an individual CPU to transition from its old owner to its new owner is therefore bounded by at most one iteration of its previous owner's period and one iteration of its new owner's period (both before the transition).

5.5 Evaluation

This section evaluates the run-time system discussed in Section 5.4. We begin by measuring overheads to gauge the efficiency of our system. We also use Kernelshark to observe the

adaptation, which shows it to be working as expected. We then also use this runtime system to show the equivalence between period-elastic and computationally-elastic tasks.

All experiments used OpenMP parallel programs written in C/C++ and compiled with GCC 4.8.2. They were run on a 32-core machine with four Intel Xeon E5-4620 processors running at a constant 2.20014 GHz with hyper-threading disabled. The RTOS is x86-64 Linux kernel version 4.1.7 with the RT-PREEMPT patch.

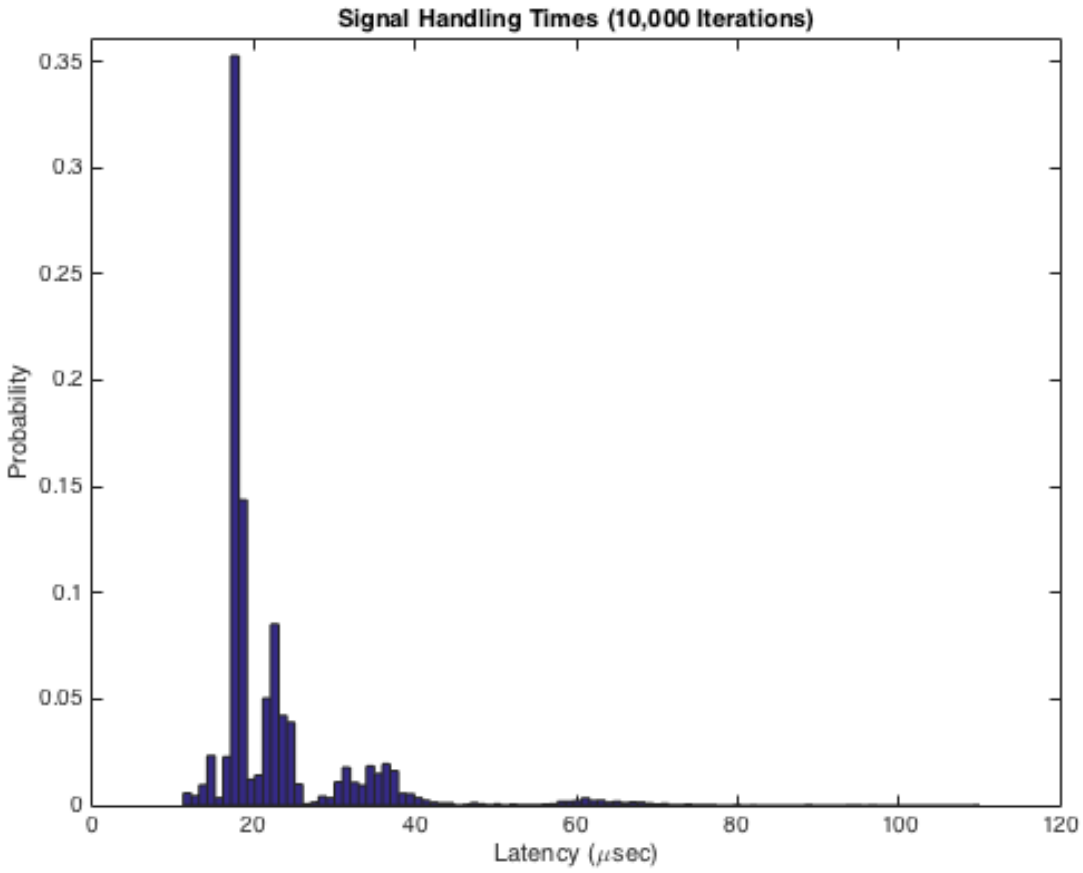


Figure 5.2: Signal Overhead Distribution

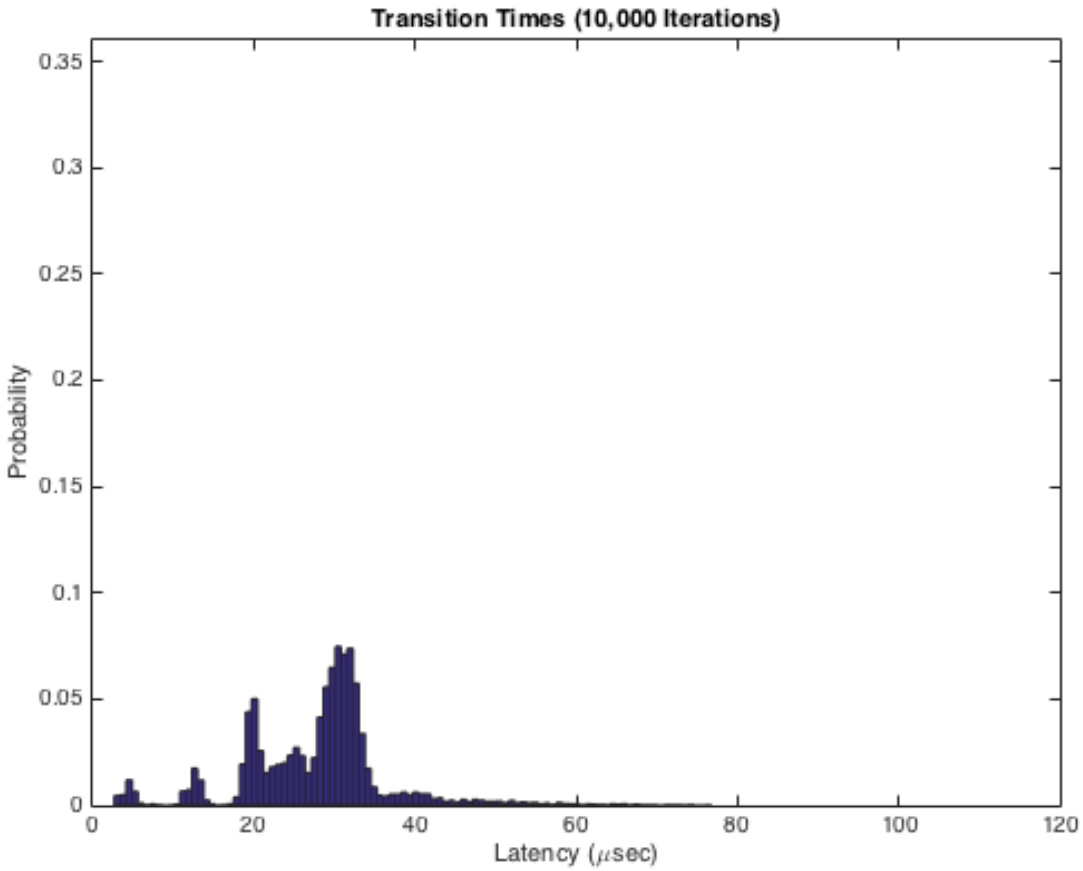


Figure 5.3: Transition Overhead Distribution

5.5.1 Overheads and Efficiency

We begin by examining the overheads of scheduling, migration, and communication mechanisms used in the runtime system.

We use POSIX RT signals to notify tasks of events. Figure 5.2 shows the distribution of the measured latency to deliver a POSIX RT signal. This experiment involved two tasks, a signal sender, and a signal receiver. The signal sender's only job was to send signals to the signal receiver. It notes the current time, sends a signal, sleeps for $250\mu sec$ and then sends another signal. The signal receiver is constantly doing busy work but is repeatedly interrupted by the signal handler. Inside the signal handler, the receiver notes the current time and returns to its busy work. This is repeated 10,000 times. The values shown in Figure 5.2 are obtained by subtracting the time recorded in the signal sender before sending the signal from the time recorded in the signal handler by the signal receiver. We note a minimum reaction time of $11.23\mu sec$ and a maximum of $110.03\mu sec$ with over 1/3 falling in the range $[18.0, 19.0)\mu sec$.

Figure 5.3 shows a distribution of the overhead associated with changing a task's real-time priority and migrating it to another processor, as must be done in the worst-case when a passive thread must change its priority and migrate to another CPU to run. This experiment consisted of a task randomly selecting a real-time priority (from 1 to 98), randomly selecting a processor (from 0 to 31) then taking note of the current time, changing its real-time priority, migrating to its new processor, and taking note of the time again. This happened 10,000 times. The times are noted in Figure 5.3. The minimum observed time was $2.67\mu sec$ (it is likely that the randomly selected CPU was the one on which the task was already running and therefore no migration was necessary–note this also happens in our system when a task obtaining a CPU already has a passive thread sleeping on the CPU it will be gaining), and the maximum observed time was $76.77\mu sec$.

5.5.2 Adaptation of a Taskset

We randomly generated several task sets in a similar fashion to prior work [35, 47]. Each task had a 50% chance of being computationally-elastic vs. period-elastic. Each task had a maximum ratio of span to minimum period of $p_{max} = \frac{1}{2(2+\sqrt{2})}$. The actual span to period ratio is first generated as a percentage of p_{max}: 40%, 50%, 70%, or 100% with probability $0.4, 0.3, 0.2$, or 0.1, respectively. Once this ratio has been determined, an actual span value is computed by repeatedly generating segments of work from a log normal distribution with mean of $5ms$ until the sum reaches the chosen percentage of one second. Each time a segment's length is chosen, it also generates a number of strands (how many times each segment must be run) from a log normal distribution with a mean of $1 + \sqrt{m}/3$. The sum of the length of each strand times its number of segments becomes the work for period-elastic tasks. For computationally-elastic tasks, two numbers of strands are generated for each segment. The lower is added to the minimum work, and the higher is added to the maximum work. For computationally-elastic tasks a period is generated uniformly between a minimum of $50ms$ and a maximum of $1s$. For period-elastic tasks two periods are generated and the higher value becomes the maximum period and the lower value becomes the minimum period. Each task's elasticity value was randomly generated over the interval (0.0, 1.0]. If at any time a task does not have a minimum of at least 2 CPUs and a maximum of at least 3 CPUs, the task is discarded and another task is generated.

We generated hundreds of task sets, and Algorithm 5 found a suitable schedule for each. For each task set we randomly selected one task that would set its period or workload (depending on the elastic nature of the task) to a randomly selected value between its minimum and maximum. All other tasks would adapt and reschedule accordingly. No deadlines were ever

missed, indicating that the transition system described in Section 5.4 indeed provides a safe and efficient reassignment of CPUs from one task to another.

Kernelshark was used to verify that tasks were behaving as they were supposed to, and after a mode change each CPU each transfer happened exactly as described in the previous section: the task giving up a CPU did so at the end of an iteration, and the task receiving the CPU would begin using it at the beginning of the following iteration. Furthermore, we saw regular periodic behavior at exactly the period expected of each task.

5.5.3 Functional Equivalence of Period–Elastic and Computationally–Elastic Tasks

To assess equivalence of period-elastic and computationally-elastic tasks we performed the following experiments: First, two identical task sets were generated with the exception of one task in each task set. In one task set the designated task is period-elastic, and in the other task set it is computationally-elastic. These two tasks are functionally equivalent in that they have the same elasticity and minimum and maximum utilization. All other tasks were generated as described in the previous subsection. We adapted the period-elastic task to run at the constant period of the computationally-elastic task. Likewise the computationally-elastic task adapted to run the constant workload of the period-elastic task. As expected the remaining tasks in each task set adapted in the same way, regardless of the designated task's elastic nature.

We show the results of four such experiments below. The period and work values before and after adaptation are shown in the charts above. For task set 1 we also plot each task in terms of its current work and period both before and after a reschedule. In the graph on the left the computationally-elastic task (triangle) and period-elastic task (square) are at different

	Work (ms)	Period (ms)	Work (ms)	Period (ms)
Task 1	2428.63	354.63	2428.63	321.71
Task 2	4283.11	914.57	3000.00	914.57
Task 3	3665.64	328.50	3665.64	328.50
Task 4	3302.41	879.40	3302.41	879.40

Table 5.1: Experiment 1 Taskset 1

	Work (ms)	Period (ms)	Work (ms)	Period (ms)
Task 1	2428.63	354.63	2428.63	321.71
Task 2	3000.00	637.53	3000.00	914.57
Task 3	3665.64	328.50	3665.64	328.50
Task 4	3302.41	879.40	3302.41	879.40

Table 5.2: Experiment 1 Taskset 2

	Work (ms)	Period (ms)	Work (ms)	Period (ms)
Task 1	2347.18	499.56	2347.18	499.56
Task 2	3975.46	643.08	4304.06	643.08
Task 3	2966.58	404.81	2000.00	404.81
Task 4	4215.17	557.39	4215.17	474.26

Table 5.3: Experiment 2 Taskset 1

	Work (ms)	Period (ms)	Work (ms)	Period (ms)
Task 1	2347.18	499.56	2347.18	499.56
Task 2	3975.46	643.08	4304.06	643.08
Task 3	2000.00	276.45	2000.00	404.81
Task 4	4215.17	557.39	4215.17	474.26

Table 5.4: Experiment 2 Taskset 2

	Work (ms)	Period (ms)	Work (ms)	Period (ms)
Task 1	5964.79	673.92	5964.79	673.92
Task 2	2895.20	354.18	2895.20	385.55
Task 3	4935.78	619.04	5000.00	619.04

Table 5.5: Experiment 3 Taskset 1

	Work (ms)	Period (ms)	Work (ms)	Period (ms)
Task 1	5964.79	673.92	5964.79	673.92
Task 2	2895.20	354.18	2895.20	385.55
Task 3	5000.00	675.65	5000.00	619.04

Table 5.6: Experiment 3 Taskset 2

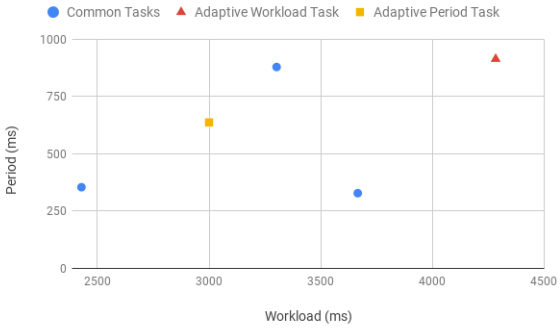
locations before the transition, but in the graph on the right, they have adapted to the star location.

	Work (ms)	Period (ms)	Work (ms)	Period (ms)
Task 1	2509.48	330.80	1750.00	330.80
Task 2	4525.07	501.44	4525.07	439.54
Task 3	3668.63	386.00	3668.63	386.00

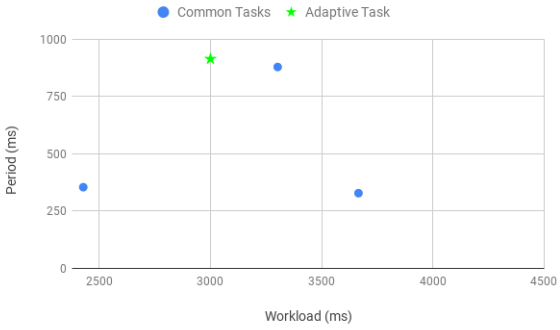
Table 5.7: Experiment 4 Taskset 1

	Work (ms)	Period (ms)	Work (ms)	Period (ms)
Task 1	1750.00	241.11	1750.00	330.80
Task 2	4525.07	501.44	4525.07	439.54
Task 3	3668.63	386.00	3668.63	386.00

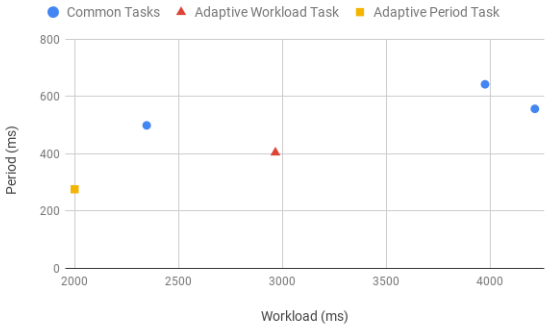
Table 5.8: Experiment 4 Taskset 2



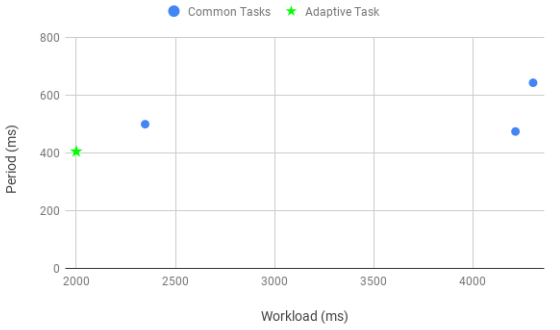
Experiment 1 Before Reschedule



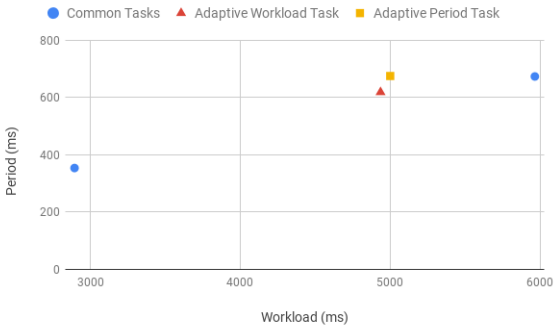
Experiment 1 After Reschedule



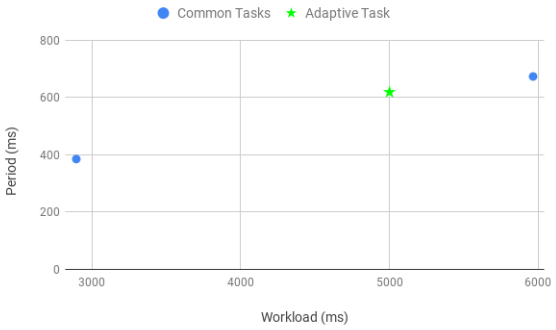
Experiment 2 Before Reschedule



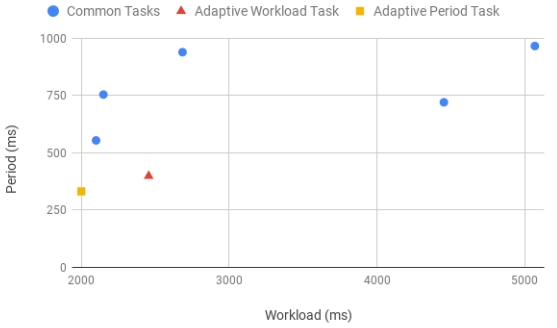
Experiment 2 After Reschedule



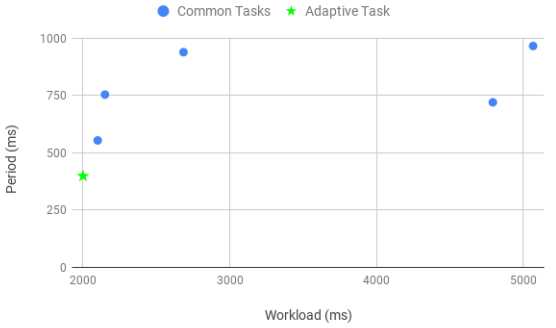
Experiment 3 Before Reschedule



Experiment 3 After Reschedule



Experiment 4 Before Reschedule



Experiment 4 After Reschedule

5.6 Summary

This chapter has extended the state of the art in elastic scheduling by introducing the concept of computational elasticity. We show that computational elasticity is functionally equivalent to period elasticity, as both can be encapsulated as utilization elasticity. We then modify existing scheduling algorithms for period-elastic tasks to include computationally-elastic tasks. We validated the equivalence of period and computational elasticity using a runtime system we built for elastic parallel real-time systems.

Chapter 6

Discrete Elastic Scheduling

This chapter introduces a new discrete model of elastic tasks in which each task potentially has multiple discrete potential "modes of operation," each with a corresponding period and worst-case execution time (WCET). The discrete nature of these tasks allows for tasks to exploit both computational and period elasticity simultaneously, which is yet unachievable[4] under the continuous elastic model. Although this task model is compatible with both parallel and sequential tasks, we again focus on the parallel case using the federated scheduling paradigm. We then adapt the runtime system presented in the previous chapter to schedule discrete elastic tasks rather than continuous ones and use it to run the first ever adaptive virtual real-time simulation experiment, which we also discuss in detail.

[4]A key assumption of the continuous elastic task model is that each task's processor utilization can vary between a minimum and maximum value. When only one of period or WCET is allowed to adapt, this corresponds to one-to-one mapping of utilization to period (or WCET). If both are allowed to change simultaneous, there are infinitely many valid period and WCET values for a given utilization.

6.1 Introduction

The ***elastic task model***, first introduced by Buttazzo et al. [7], allows for online modification of task periods to maintain schedulability of adaptive ***period-elastic*** tasks without the pessimism required for a static schedule accommodating the worst-case behavior of the most utilization-intensive mode of operation. That model was later extended to include both tasks with internal parallelism and tasks that instead can adapt their computational loads (***computational elasticity***). [42, 43]

We provide a new elastic task model in this chapter that further expands the state of the art by introducing ***discrete elastic scheduling*** in which each task's assigned utilization is obtained from a finite set of candidate tuples, each of which has an associated period and workload. From one mode of operation to the next, a task may change its period, its computational workload, or ***both***. The discrete elastic model more accurately describes tasks that have distinct modes of operation, such as a robot with multiple available planning algorithms with varying degrees of computational demand, or a control application that may get better results from running at a higher frequency but needs to maintain harmonic rates with respect to other tasks in the system. Unlike the continuous elastic model, the discrete model allows adaptation of both computational demand and period together, at once (***combined elasticity***).

We use the real-world application domain of ***real-time hybrid simulation (RTHS)***, used by earthquake engineers to understand structural behavior with high fidelity at realistic time-scales [18, 20, 44], as a motivating example for discrete elastic scheduling. In RTHS a well-understood portion of a structure is simulated while a portion to be tested or validated is physically built. The combined structure is then connected via sensors and actuators and subjected to external stimuli (such as earthquake ground motions) at fine-grained time scales

in order to examine how the relevant portions behave. Different portions of the structure can be simulated at different rates to yield resources to portions of special interest (e.g., those near the physical specimen). However, to date, resources have been statically assigned in RTHS experiments: each substructure runs at a fixed rate with a fixed set of computational resources, and changes to the system can only be made between successive runs. We exploit discrete elastic scheduling to conduct the first ever (virtual) real-time hybrid simulation experiment in which resource adaptation enables adaptive switching between controllers with different computational demands. In this experiment, the control algorithm that determines the response to the system's behavior is able to execute in multiple modes of operation, i.e., using a non-linear Kalman filter vs. a more computationally-expensive particle filter. Other tasks in the system (which must run at rates harmonic with that of the control algorithm) are similarly able to adapt their periods, computational loads, or both, accordingly.

This chapter is structured as follows. Section 6.2 provides relevant background information. Section 6.3 presents the ***discrete elastic scheduling system model***, including a discussion of the implications of ***combined elasticity***, which allows for tasks to adapt both computational workload *and* period. In Section 6.3 we also prove the scheduling of parallel tasks using this model under the federated scheduling paradigm to be (weakly) NP-hard via a reduction from the Knapsack Problem. We then present a pseudo-polynomial time dynamic-programming algorithm (obtained by reducing our scheduling problem to an instance of the Multiple Choice Knapsack Problem) that can efficiently create an optimal schedule for such tasks. Section 6.4 describes our adaptive virtual RTHS experiment. Section 6.5 evaluates the level of pessimism when using discrete elastic scheduling vs. idealized (but often practically unsuitable) continuous elastic scheduling. Section 6.6 summarizes the chapter and describes future directions for extending this work.

Although this chapter focuses on the discrete elastic scheduling of parallel real-time tasks under federated scheduling, we point out that many of the concepts introduced here are also applicable to sequential tasks as is discussed in Footnote 5 in Section 6.3; hence, our proposed model should be considered an extension of the elastic task models for sequential *and* parallel workloads.

6.2 Background

In this chapter we present the novel concept of *discrete elastic scheduling*, focusing on discretely elastic parallel real-time tasks under the federated scheduling paradigm. This section provides background information about the example application domain that motivates our approach and is used to evaluate it: *real-time hybrid simulation (RTHS)*.

6.2.1 Elastic Scheduling

As was discussed extensively in earlier chapters of this dissertation, in the continuous parallel period-elastic task model, each task is formally represented as $\tau_i = \left\langle C_i, L_i, T_i^{(max)}, T_i^{(min)}, E_i \right\rangle$ where C_i represents the task's constant *worst-case execution time (WCET)* on a single processor, L_i is the WCET on an infinite number of processors, and the closed range $[T_i^{(min)}, T_i^{(max)}]$ spans all acceptable period values for a task, where a lower period (and therefore higher utilization) is always preferred. The *current* period is denoted T_i. A task's *elasticity coefficient* E_i is a measure of how relatively easy or difficult it is to change a task's period.

The ***federated scheduling*** paradigm was first introduced by Li et al. [36] to schedule sporadic parallel tasks represented as ***directed acyclic graphs (DAGs)***, each with a utilization $U_i \geq 1$ that demands more than a single processor. These ***high-utilization***

tasks are each given exclusive use of m_i processors according to the equation

$$m_i = \left\lceil \frac{C_i - L_i}{T_i - L_i} \right\rceil \tag{6.1}$$

while low-utilization tasks are scheduled sequentially.

This dissertation extended the elastic task model to include *parallel real-time DAG tasks* under federated scheduling. To keep parallel elastic scheduling as semantically equivalent to Buttazzo's original model as possible, we presented an optimal scheduling algorithm that directly solves a minimization problem similar to that given in Equation 5.1:

$$\textbf{minimize} \sum_{i=1}^{n} \frac{1}{E_i}(U_i^{(max)} - U_i)^2 \tag{6.2}$$

such that:

$$U_i^{(min)} \leq U_i \leq U_i^{(max)} \text{ for all } \tau_i$$

and

$$\sum_{i=1}^{n} m_i \leq m$$

Each task is initially given its minimum number of processors, and the remaining CPUs are allocated in a manner that minimizes the sum in Equation 6.2.

Noting that task utilization is dependent on *both computational load and period*, this allows for tasks to have a range of acceptable utilizations [$U_i^{(min)}$,$U_i^{(max)}$] that can be either a range of acceptable periods [$T_i^{(min)}, T_i^{(max)}$] as in Buttazzo's model ***or*** a range of acceptable computational loads [$C_i^{(min)}, C_i^{(max)}$]. Tasks that adapt their periods are called ***period-elastic tasks***, while tasks that adapt their workloads are ***computationally-elastic tasks***. This chapter further extends that elastic task model, (1) allowing for the more realistic scenario of discrete candidate utilization values instead of continuous ranges; and in doing so also (2)

allowing for ***combined-elastic tasks*** to adapt both their periods and computational loads at once.

6.2.2 Motivating Application Domain

Although the adaptive capabilities and discrete workloads enabled by discrete elastic scheduling are relevant to a variety of real-time applications, we focus here on ***real-time hybrid simulation (RTHS)***, which is used by structural engineers to study the dynamic behavior of a structural specimen under loading that potentially results in unknown and highly nonlinear behavior. Traditionally, a new structural concept or a new vibration mitigation device was validated in one of two ways: a physical structure was built and subjected to tests, or a numerical model was tested via computer simulations. However, building physical structures, even if not at full scale, and subjecting them to full physical tests, though robust, can be prohibitively expensive in terms of money and time. On the other hand, running computer simulations such as finite element models is less expensive but may not fully capture nuances of a physical structure: for instance, accurate numerical models may not exist for some types of damage that a physical structure could sustain.

Hybrid simulation combines the strengths of purely physical and purely numerical approaches. A portion of a structure is physically built to be studied, while the remainder is simulated numerically. The complete structure (composed of both physical and simulated components) is then dynamically subjected to external loads (such as earthquake ground motions) during experimentation, resulting in a feedback control system with numerical models that must be executed on-line. The physical components are driven by actuators, and their displacement, velocity, and acceleration are measured by sensors and input back into the computational subsystem. The resulting computation in turn determines the forces the actuators should apply to the physical substructure in the next time step.

Hybrid simulation is often done at rates that are too slow to evaluate the dynamic performance of rate-dependent structural systems. This can be remedied by running the system at fine-grained time scales with real-time requirements, as a *real-time hybrid simulation (RTHS)* [18, 20]. A widely-used platform for RTHS is MathWorks's Speedgoat/XPC Target that runs in coordination with real-time Simulink. However, such a system is neither parallel nor adaptive, which limits the kinds of experiments that it can run.

The potential for extensive damage to equipment, test specimens, or even people as a result of unintended actuation in the case of an unstable control algorithm necessitates that before full RTHS experimentation can be done safely, as much validation of the proper system setup as possible must be performed. One such validation that always precedes a RTHS is a ***virtual RTHS*** in which the physical component of RTHS is replaced by a simulation, often on an entirely different machine and using the same interface as the physical component. Although the simulated "physical component" in virtual RTHS cannot fully capture the dynamics of the actual physical specimen under examination in full RTHS (indeed the partially unknown dynamics of the physical specimen may be the very reason for running the RTHS experiment), a virtual RTHS can effectively validate control algorithms and numerical models that will be used in RTHS experiments. As such, in this chapter we present an adaptive virtual RTHS using discrete elastic scheduling in Section 6.4) as a crucial first step towards adaptive RTHS.

Multi-time-stepping (MTS) decomposes an RTHS into subsystems (with individual tasks) and runs each task at its own harmonic periodic rate, where for any two subsystems, the periodic rate of one has a ***time-step ratio*** of x times that of the other. Data is exchanged at each iteration of the slower of the tasks to ensure subsystems have a consistent view of the overall system. Multi-time-stepping allows for more precise control over individual subsystems' periods (e.g., one subsystem runs relatively quickly in order to read a vital physical sensor more frequently or another subsystem runs more slowly in order to process

more simulation data in each period) than if the entire system were running at a single periodic rate. However, multi-time-stepping alone does not allow for fine-grained control over tasks' computational loads. Nor does it allow for run-time re-allocation of resources (e.g., which would allow for a subsystem's runtime behavior to change with its workload) [6].

The Cybermech platform was developed by Ferry et al. [18] to run parallel RTHS experiments via federated scheduling. Although Cybermech supports multi-time-stepping, each subsystem only runs at a fixed periodic rate [6], and thus is only applicable to systems whose control model is linear. Similarly, recent work [46] has formulated algorithms and heuristics for the non-preemptive multi-processor scheduling of RTHS systems (with multi-time stepping) based on Functional Mocked-up Interface (FMI) diagrams using integer linear programs. This system aso uses static resources.

In contrast, the discrete elastic scheduling approach introduced in this chapter allows for dynamic re-allocation of individual subsystems' periodic rates and/or computational resources to accommodate linear and potentially non-linear behavior which can occur with new, experimental energy-damping devices. We demonstrate such adaptive resource management capabilities and use them to enable adaptive switching between controllers with differing computational demands for the first time in a virtual RTHS as is described in Section 6.4.

6.3 Discrete Elastic Scheduling

In this section we present the discrete elastic task model for parallel real-time systems. We then discuss implications of the *combined-elastic* adaptations enabled by this model. We also prove that scheduling of discrete elastic tasks under federated scheduling is NP-Hard in the weak sense, and provide a pseudo-polynomial time algorithm for scheduling such tasks. [5]

[5]In this chapter we focus on scheduling high-utilization tasks ($U_i \geq 1$) via federated scheduling, although low-utilization tasks ($U_i < 1$) can be scheduled sequentially on a uniprocessor in a fashion similar to that

6.3.1 Task Model

Similar to the continuous elastic task model, in the discrete elastic task model, each task τ_i has elasticity coefficient E_i and the assigned utilization U_i of each task can range between $U_i^{(min)}$ and $U_i^{(max)}$. However, in the discrete model, rather than allowing any utilization within the continuous range $[U_i^{(min)}, U_i^{(max)}]$, each parallel task τ_i has exactly k_i discrete modes of operation. Each mode of operation $j (1 \leq j \leq k_i)$ for each task has a specific period (and implicit deadline) $T_i^{(j)}$, work $C_i^{(j)}$, and span $L_i^{(j)}$. The candidate utilizations for the task come from the period and work in each of these modes of operation $U_i^{(j)} = C_i^{(j)}/T_i^{(j)}$, and $U_i^{(min)}$ and $U_i^{(max)}$ are the lowest and highest such utilizations, respectively. In period-elastic tasks, all modes have the same work and span values (i.e., $\forall x, y; 1 \leq x \leq k_i, 1 \leq y \leq k_i; C_i^x = C_i^y, L_i^x = L_i^y$). Similarly, all modes of operation in computationally-elastic tasks have the same period (i.e., $\forall x, y; 1 \leq x \leq k_i, 1 \leq y \leq k_i; T_i^x = T_i^y$). We use Equation 6.1 to determine $m_i^{(j)}$, the number of processors required to schedule τ_i in mode j. [6]

We seek to schedule n tasks on m processors by selecting a mode of operation j for each task τ_i $(1 \leq j \leq k_i)$ while minimizing Equation 6.2. A pseudo-polynomial time algorithm for this is presented in Section 6.3.4.

---

described in this section by focusing on keeping their aggregate system utilization of each selected tuple below a desired utilization U_d rather than counting the number of processors that have been allocated. We further restrict the task model to assume that no tasks can transition from low-utilization to high-utilization (or vice-versa) when elastically re-scheduling. I.e., for each task τ_i ($U_i^{(min)} \geq 1$ and $U_i^{(max)} \geq 1$) or ($U_i^{(min)} < 1$ and $U_i^{(max)} < 1$). The scheduling of tasks whose utilization range spans both high and low values is a more difficult problem that we leave for future work.

[6] We assume each task receives at least 1 dedicated CPU under federated scheduling.

Uniqueness

On its face the discrete elastic task model presented in this chapter is similar to one used by Kuo and Mok [32] to model adaptive real-time tasks decades ago. However, there are several key differences.

Both models have a set of adaptive tasks with candidate modes of operation. However, whereas our model allows for arbitrary C_i and T_i combinations between modes of operation, the model presented by Kuo and Mok scales task periods and workloads under a constant utilization. For instance, $\tau_i = (C_i, T_i)$ may have candidate modes $(2, 4), (2.5, 5), (3, 6)$ in [32] where all modes necessarily have a utilization of 0.5. This is allowed in the discrete elastic model presented here. However, a fourth candidate mode of $(2, 5)$ with utilization 0.4, which is also perfectly acceptable in our model, is not allowed in theirs.

Furthermore [32] seeks to assign periods in such as way as to maximize harmonic chains and therefore maximize schedulability on a uniprocessor. The period-assignment problem asks whether there is a period assignment such that the maximum harmonic base is at least a certain value. This problem is proven to be strongly NP complete (i.e. no pseudo-polynomial time algorithm exists unless P=NP). The problem considered in this chapter is fundamentally different. This model does not (necessarily) care about harmonic chains and uses a pseudo-polynomial time dynamic programming algorithm for period selection.

6.3.2 Discussion

The continuous elastic task model allows for tasks to adapt their periods *or* their workloads to any value over a continuous range depending on the needs of the system. This is useful for many kinds of tasks. Consider, for instance, an "anytime algorithm" [14] that can return a

valid answer at any instant with the quality of the answer improving as the algorithm is allowed to run longer. Such an algorithm can be modeled as a task with an elastic computational requirement that may vary over a continuous range. However, not all algorithms are anytime algorithms: for some tasks, meaningful results are only returned if the algorithm is allowed to execute for certain specific durations. In a similar vein, periodic tasks that form part of a control loop may need to execute at frequencies (and hence period values) that are consistent with the remainder of the control loop (e.g., harmonic with respect to the base system frequency), and cannot operate with arbitrary periods.

Therefore, the continuous elastic task model is not appropriate for some important kinds of tasks. This becomes more apparent when one considers that on actual hardware, task execution times are essentially discrete. Processors treat time not as a continuous interval but as a discretized count of cycles. Therefore on a general-purpose CPU, no job can actually run for an arbitrary amount of time, but instead executes for an integer number of CPU cycles.

Under the discrete elastic task model, each task τ_i has k_i unique modes of operation, each of which has an associated period and workload. Varying only a single dimension (i.e., changing only the period or workload as in the continuous model) may allow for more appropriate management of the selected attribute than the continuous elastic model. For instance, the discrete elastic task model allows for the guaranteed selection of harmonic periods among period-elastic tasks.

Perhaps an even greater benefit of the discrete elastic model is its ability to allow exploitation of *both period elasticity and computational elasticity*. This ***combined elasticity*** increases the range of potential modes of operation for a given task. Figures $6.1 - 6.4$ demonstrate the diversity of adaptations enabled by combined elasticity. Each of the four images shows the same task exploiting different types of elasticity. The y-axis is the task's computational

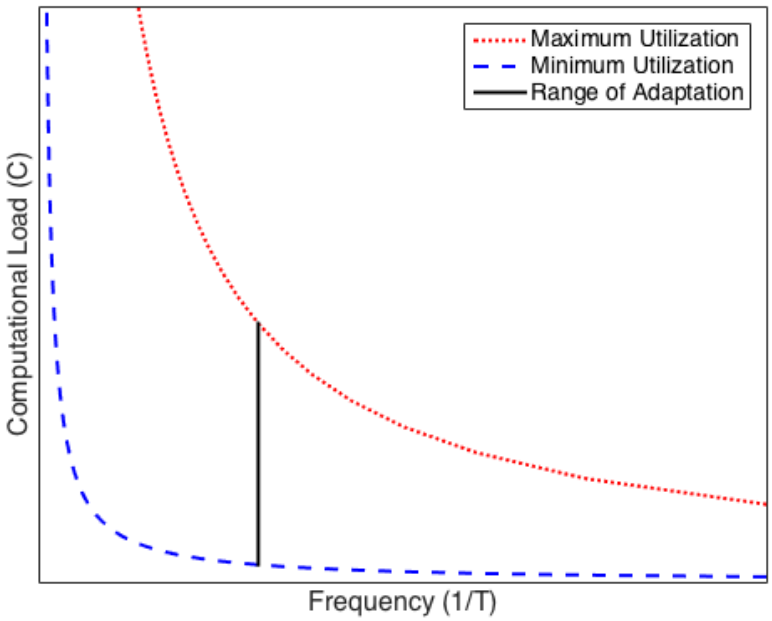


Figure 6.1: Continuous Computationally-Elastic Task

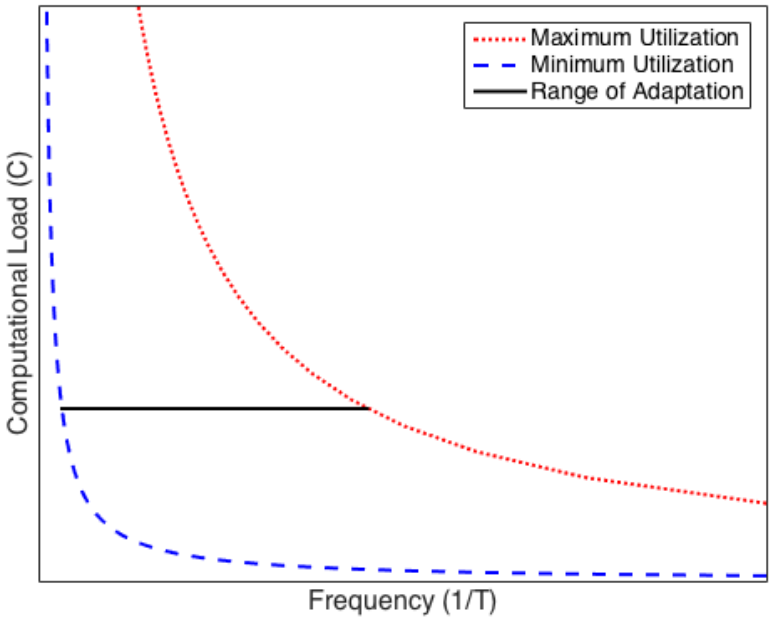


Figure 6.2: Continuous Period-Elastic Task

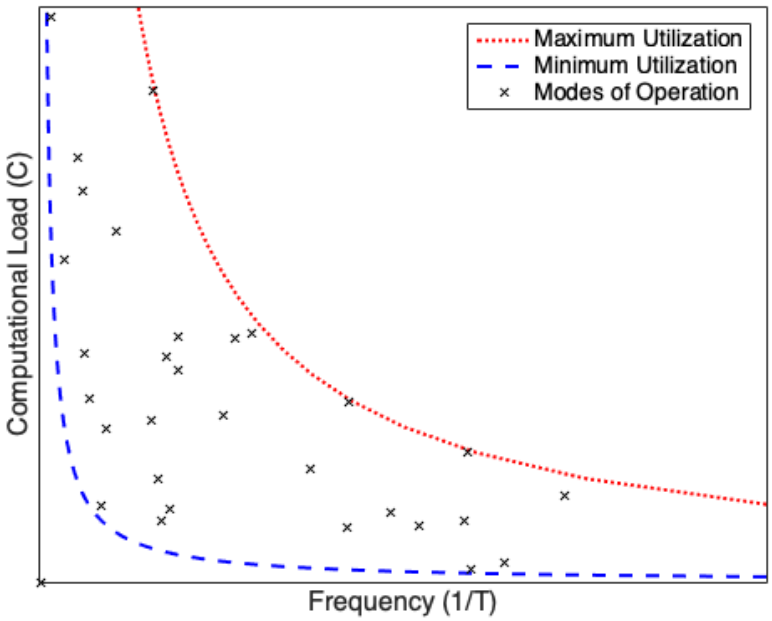


Figure 6.3: Discrete Combined-Elastic Task

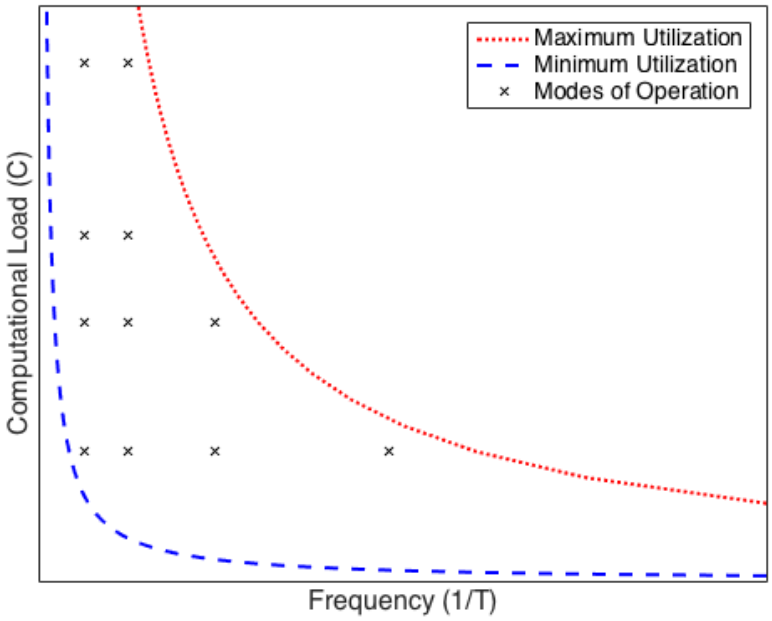


Figure 6.4: Discrete Workloads and Harmonic Rates

load (C), and the x-axis is its frequency ($1/T$). Any point within the allowed region therefore represents a potential work and period assignment for the task. Constant values $U^{(min)}$ and $U^{(max)}$ are represented by dashed and dotted curves, respectively, so any valid assignment of C and T must therefore fall between these two curves.

Figures 6.1 and 6.2 show the potential period and workload values of a computationally-elastic task and a period-elastic task, respectively, under the continuous elastic task model. Although there are an infinite number of acceptable period (or workload) values that keep the utilization between $U^{(min)}$ and $U^{(max)}$, the range of adaptation for a single task is relatively narrow.

Contrast this with Figure 6.3, which demonstrates the potential period and workload values of combined-elastic tasks enabled by the discrete elastic task model. Although there are finitely many of modes of operation, adaptation is allowed in both computational and period dimensions, allowing for a much broader adaptation space. Any point in the entire region between the minimum and maximum utilization curves may be a candidate mode of operation. Thirty such (randomly-chosen) points are plotted in Figure 6.3.

Which (and how many) candidate points are available is then a configurable application-specific concern. Anytime tasks that can perform arbitrary amounts of work for arbitrary time periods have an arbitrary number of possible period and workload candidates which can be selected from anywhere between the minimum and maximum utilization curves (subject to inherent discretization of work by the processor, etc.). System designers then can select as many or as few potential modes of operation as appropriate.

In other cases, application constraints such as the need to run at harmonic rates or a fixed set of computational completion points may restrict or even determine actual modes of operation. Furthermore, in some applications, like real-time hybrid simulation (RTHS) for example, adaptation of both period and computational workload may be useful since many of its

sub-structures can adapt in either or both dimensions. Depending on the computational resources available to it, an RTHS task may simulate a substructure in more or less detail (thereby having different computational loads). It may similarly run its simulation at a faster or slower harmonic rate (as constrained by inter-task data dependencies and by the rate of the RTHS control feedback loop). Figure 6.4 shows a sample RTHS task with four potential harmonic periods and four potential workloads. Note that as Figure 6.4 illustrates, not all workloads can be run at all harmonic periods since the utilization may exceed the maximum utilization curve as the workload increases and the period decreases.

Finally, we note that some loss of utilization may be incurred by discretization. For instance, if the same period-elastic task were scheduled under both the continuous and discrete elastic models, the continuous model may assign a task a feasible period that is between two discrete candidate periods. To maintain schedulability, the task may need to be assigned the longer of the two periods under the discrete model, thereby resulting in a lower utilization than under the continuous model. However, we note that the smaller the gap between candidate periods in the discrete model, the smaller the loss of such system utilization due to discretization is. Anytime tasks can exploit this small loss of utilization by selecting many potential modes of operation that are close together in both dimensions, to approximate continuous elasticity while gaining the benefit of combined elasticity, at a (potentially acceptable) cost of a longer-running scheduling algorithm (see Section 6.3.4). We discuss and study potential utilization loss due to discretization further, in Section 6.5.

6.3.3 Proof of NP-Hardness

We now prove that the federated scheduling of parallel discrete elastic tasks is NP-hard, via a reduction of an instance of the Knapsack Problem [29] to an instance of the Discrete Elastic Scheduling Problem.

Theorem 1. *Discrete Elastic Scheduling is NP-hard.*

Proof: Reduce KNAPSACK to Discrete Elastic Scheduling.

An instance of KNAPSACK is specified as follows:[7]

$$I_{\text{KNAPSACK}} = \left\langle \left\{ (s_i, v_i \right\}_{i=1}^{n}, S, V \right\rangle$$

where the objective is to fill a knapsack of capacity S with items chosen from a set of n items, and item i ($i = 1...n$) has *weight* s_i and *value* v_i, such that the weight of the selected items sum to no more than the knapsack's capacity S and their combined value is maximized, with a total of at least the target value V.

Given such a specification, we construct an instance of the Elastic Scheduling problem with n tasks, each of which has 2 modes of operation, to be scheduled on $(n + S)$ processors. All n tasks have the same period in all modes of operation, denoted x (i.e., all tasks are computationally-elastic–we note that though all tasks in this construction are computationally-elastic, the same algorithm also schedules period-elastic and combined-elastic tasks). We first construct each task's first mode of operation as follows: Assign $C_i^{(1)} = L_i^{(1)} = x$ for all i. As a consequence these are all sequential zero-slack modes of operation, and $m_i^{(1)} = 1$ (for all i). For each i, define the second mode of operation as $C_i^{(2)} = x \cdot (1 + s_i)$ and $L_i^{(2)} = 0$. These are "embarrassingly parallel" modes of operation. Note that we consequently have $m_i^{(2)} = (1 + s_i)$. Let elastic coefficient $E_i = s_i^2/v_i$.

Note that choosing the second mode of the i'th task requires an additional s_i processors (since the first mode requires 1 processor). Let Γ_1 and Γ_2, respectively, denote the tasks for which the first mode and second mode, respectively, are selected. Recall that in Elastic

[7] All parameters are assumed to be rational numbers.

Scheduling, we seek to minimize $\sum_i \frac{1}{E_i}\Big(U_i^{(\max)} - U_i\Big)^2$. Therefore:

$$\sum_i \frac{1}{E_i}\Big(U_i^{(\max)} - U_i\Big)^2$$

$$\equiv \sum_i \frac{1}{E_i}\Big(\frac{C_i^{(2)}}{x} - U_i\Big)^2$$

$$\equiv \sum_i \frac{1}{E_i}\Big(\frac{x \cdot (1+s_i)}{x} - U_i\Big)^2$$

$$\equiv \sum_{i\in\Gamma_1} \frac{1}{E_i}\Big((1+s_i) - U_i\Big)^2 + \sum_{i\in\Gamma_2} \frac{1}{E_i}\Big((1+s_i) - U_i\Big)^2$$

$$\equiv \sum_{i\in\Gamma_1} \frac{1}{E_i}\Big((1+s_i) - 1\Big)^2 + \sum_{i\in\Gamma_2} \frac{1}{E_i}\Big((1+s_i) - (1+s_i)\Big)^2$$

$$\equiv \sum_{i\in\Gamma_1} \frac{s_i^2}{E_i}$$

$$\equiv \sum_{i\in\Gamma_1} v_i$$

We thereby conclude that a solution to the Discrete Elastic Scheduling Problem in which the function in Equation 6.2 takes on a value at most

$$\Big(\sum_i v_i\Big) - V$$

exists if and only if $I_{\text{KNAPSACK}} \in$ KNAPSACK. ■

6.3.4 Pseudo-Polynomial Time Scheduling Algorithm

MULTIPLE-CHOICE KNAPSACK [48] is similar to KNAPSACK, but rather than selecting items from a single set, there are multiple mutually-exclusive sets, and exactly one item must be chosen from each set in such a way as to maximize profit and ensure a total weight below the

Algorithm 6 Multiple Choice Knapsack Elastic Scheduling (MCKES)

1: $MCKES[0][l] \leftarrow \infty$
2: $MCKES[d][0] \leftarrow \infty$
3: **for** $d \leftarrow 1...m$ **do**
4: **for** $l \leftarrow 1...n$ **do**
5: $MIN \leftarrow \infty$
6: **for** $j \leftarrow 1...k_l$ **do**
7: **if** $d - m_l^{(j)} \geq 0$ **then**
8: **if** $l == 1$ ***and*** $\frac{1}{E_l}(U_l^{(max)} - U_l^{(j)})^2 < MIN$ **then**
9: $MIN \leftarrow \frac{1}{E_l}(U_l^{(max)} - U_l^{(j)})^2$
10: **else if** $MCKES[d - m_l^{(j)}][l-1] + \frac{1}{E_l}(U_l^{(max)} - U_l^{(j)})^2 < MIN$ **then**
11: $MIN \leftarrow MCKES[d - m_l^{(j)}][l-1] + \frac{1}{E_l}(U_l^{(max)} - U_l^{(j)})^2$
12: **end if**
13: **end if**
14: **end for**
15: $MCKES[d][l]$ $\min(MIN, MCKES[d-1][l])$
16: **end for**
17: **end for**
18: **return** $MCKES[m][n]$

knapsack's capacity. We now provide a pseudo-polynomial time algorithm for Discrete Elastic Scheduling by reducing an instance of it to an instance of Multiple-Choice Knapsack.

A pseudo-polynomial time algorithm. We define the following reduction from Discrete Elastic Scheduling to Multiple-Choice Knapsack: each of the n tasks with k_i modes of operation becomes one of n mutually exclusive sets with k_i distinct items. Each task in Discrete Elastic Scheduling needs a mode to be selected, and each set from Multiple-Choice Knapsack needs one item to be selected. Task τ_i operating in mode j becomes an item in the corresponding set with profit $\frac{1}{E_i}\left(U_i^{(\max)} - U_i^j\right)^2$ and weight $m_i^{(j)}$. The knapsack has capacity m. By giving each item weight $m_i^{(j)}$, we ensure that if they fit in a knapsack of capacity m, then the corresponding tasks in the selected modes are schedulable on m processors. Although traditional Multiple-Choice Knapsack seeks to maximize the value of selected items, we instead attempt to minimize the value in Equation 6.2, which is exactly the profit assigned to each item. We thus use a min() function in place of a max() function that would otherwise be used, which has no bearing on the correctness or complexity of the algorithm.

We note that by successfully selecting one item from each mutually-exclusive set for the knapsack while keeping their combined weight within the knapsack's capacity m, we also select a mode of operation for each task on at most m processors. We therefore have a valid parameterization of the Discrete Elastic Scheduling instance.

A pseudo-polynomial dynamic programming algorithm presented in [30] finds an optimal solution to Multiple-Choice Knapsack by considering the maximum value achievable when considering the first l mutually exclusive sets and reduced knapsack capacity d, in our case, $1 \leq l \leq n$ and $1 \leq d \leq m$. We reproduce a slightly modified version of this algorithm

in Algorithm 6: rather than finding the maximum "value" of items in a knapsack, we seek to minimize $\sum_{i=1}^{n} \frac{1}{E_i}(U_i^{(max)} - U_i)^2$.

In Algorithm 6 we build a two-dimensional table *MCKES* where *MCKES[d][l]* gives the optimal solution after considering the first l tasks on d processors. We begin by assigning a score of infinity (since we are minimizing) to both the impossible case of scheduling l tasks on 0 processors (Line 1) and the trivial case of scheduling 0 task on d processors (Line 2).

The *for loop* beginning on Line 3 considers scheduling tasks on d CPUs. The inner *for loop* beginning on Line 4 similarly considers the first l tasks on the d processors available. While iterating we assign each task a mode of operation, with the goal of minimizing the objective function in Equation 6.2. Hence we assign the MIN score of each task an initial score of infinity (Line 5) and consider each mode j of operation in turn (Lines 6-14). Line 7 makes sure there are enough unallocated processors to select mode j. If not, we disregard mode j. Otherwise, we consider whether selecting mode j decreases the current minimum (Line 10). If so, the new minimum value is stored (Line 11). In the special case that $l == 1$ (this is the first task scheduled), the MIN score simply becomes $\frac{1}{E_l}(U_l^{(max)} - U_l^{(j)})^2$ (Lines 8-9). After considering all potential modes of operation, we assign *MCKES[d][l]* the minimum of MIN and *MCKES[d − 1][l]* (Line 15). The final optimal value is found at *MCKES[m][n]*. One can keep track of which mode is selected at each iteration for task τ_l, and the set of modes that give the value in *MCKES[m][n]* are then assigned to the their respective tasks.

Runtime complexity. Algorithm 6 has worst-case running time $\Theta(m \times N)$, where $N = \sum_{i=1}^{n} k_i$, as there are m CPUs to allocate (*for loop* beginning on Line 3) and N modes of operation selected for each value of m (*for loops* beginning on Line 4 and on Line 6).

A note about sequential tasks. As alluded to in Footnote 5 in Section 6.3, Algorithm 6 can be applied to the scheduling of sequential elastic tasks on a uniprocessor by: (1) assigning

each item associated with a candidate mode of operation, a weight equal to the corresponding utilization; and (2) assigning the knapsack a capacity equal to the desired system utilization U_d.

6.4 Adaptive Virtual Real-Time Hybrid Simulation Experiment

To evaluate our discrete elastic scheduling approach and to validate its usefulness in a real-world application, in this section we present a virtual RTHS experiment that (1) has tasks with various discrete work and period values in different modes of operation, (2) can exploit our discrete elastic scheduling approach at run-time to improve experiment accuracy by switching adaptively between modes of operation, and (3) can handle constraints like harmonic rates and discrete workloads effectively. To our knowledge, this is the first time even a virtual RTHS that can adapt its period and/or computational load has been conducted.

This simple experiment is meant as a proof of concept that discrete elastic scheduling and the adaptations thereby enabled are beneficial to real-world applications (namely RTHS). Therefore, we start with a less complicated setup than would be involved with validating a new structural component. This virtual RTHS is a tracking problem, meaning we send a displacement signal to a moving non-linear spring (henceforth referred to as the *plant*), and we attempt to make the plant follow the displacement given in the input signal as closely as possible.

The details of our experiment are shown in Figure 6.5. The input into the system is a recording of the displacement of a physical specimen that has been excited by forces taken from the El Centro earthquake. This is sent to an inverse compensator, which enhances

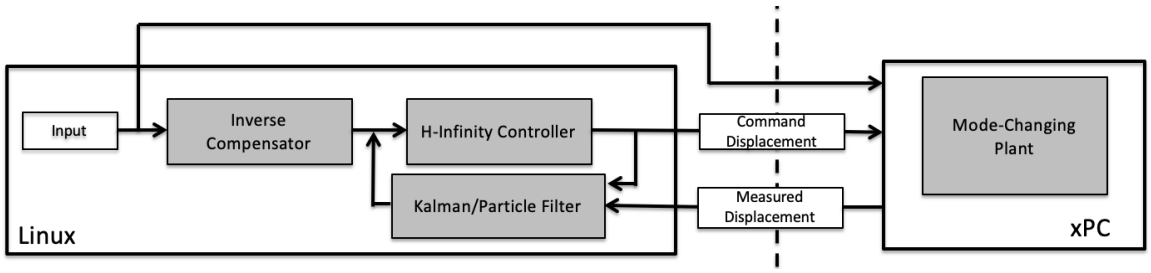


Figure 6.5: Virtual RTHS Details

tracking performance by reducing/smoothing small residual time delays introduced by the control algorithm. The controller itself uses a modified robust integrated actuator control (RIAC) strategy [44], which uses H-infinity optimization [24] to provide a trade-off between performance and robustness. The H-infinity controller uses the smoothed desired displacement passed to it from the inverse compensator and an estimate of the plant's current location to determine a command displacement to send to the plant. This estimate is the output of either a Kalman filter or a particle filter (depending on which mode of operation the task is in), both of which provide an estimate of the plant's current displacement based on noisy data (the last known measured displacement of the plant and the last commanded displacement). Each of these is calculated once per iteration and both inform the behavior of the system in the next iteration. It is assumed that when the desired displacement exceeds a certain threshold (i.e., when the plant is too far from its origin), the plant's behavior becomes more difficult to predict. Therefore, the more computationally-expensive particle filter is used then, while the Kalman filter is used otherwise.

All of the above components except the plant (which is simulated on an xPC target machine[8]) are run within a single task on Linux with the RT-PREEMPT patch. The relative simplicity of this experiment means that multiple tasks are not needed to accomplish the main goal of this virtual RTHS experiment. To gauge our approach more fully however, for scenarios where

[8]MathWorks's Speedgoat/XPC Target runs in coordination with real-time Simulink. It is a widely-used platform for a variety of cyber-physical systems, including RTHS.

there may be different substructures of a building to simulate (at potentially different rates or detail levels) within a realistic structural validation, we generate additional synthetic discrete elastic tasks to run alongside the vRTHS task, as there would be in a more complex virtual RTHS. These tasks adapt with respect to the virtual RTHS task whenever it changes from using the Kalman filter to the particle filter (or vice versa). Similar to a structural validation RTHS experimentation with multi-time stepping, we constrain each synthetic task to run at a rate that is harmonic with the 2048Hz rate needed by the virtual RTHS. Some of these new tasks are period-elastic, some are computationally elastic, and some are combined-elastic.

To perform this experiment, we extended the parallel (continuous) elastic concurrency platform from [42], which is available as open-source [45]. The underlying system calls, concurrency mechanisms, and synchronization techniques remain unchanged, but we replaced the original scheduling algorithm with Algorithm 6. All tasks were run on a 16 core machine with two Intel E5-2687W processors running at a constant 3092.616 MHz with Hyperthreading disabled. The RTOS used was x86-64 Linux with the RT-PREEMPT patch, and all programs were written in C++ and compiled with GNU G++ 5.2.0.

Figures 6.6 and 6.7 show the results of our adaptive virtual RTHS experiment. The solid line shows the curve of the desired plant displacement, while the dotted line shows the estimated displacement output from the particle filter or the Kalman filter. The horizontal lines mark the mode-change criterion. For any desired displacement between the lines, the estimator uses the Kalman filter. The system switches modes and uses the particle filter when the plant's desired displacement is too far from its origin, i.e., outside the lines.

Looking at Figure 6.6, the two curves appear nearly indistinguishable. However, when we zoom in on the peaks in Figure 6.7, the difference becomes visible.

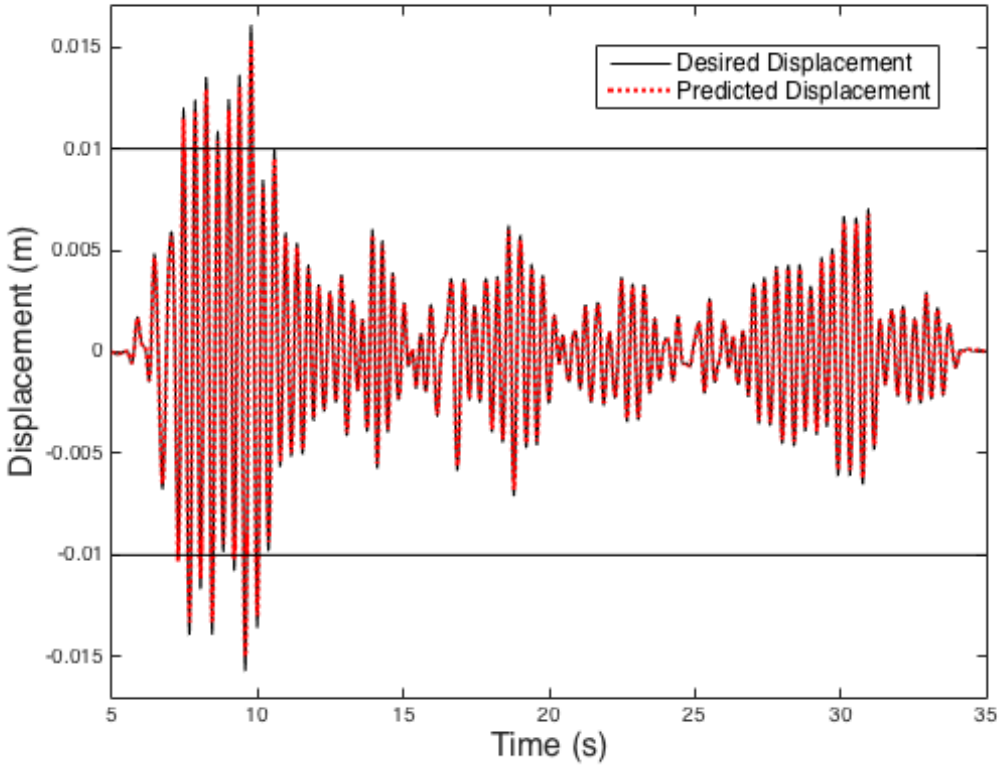


Figure 6.6: vRTHS Desired vs. Predicted Displacement

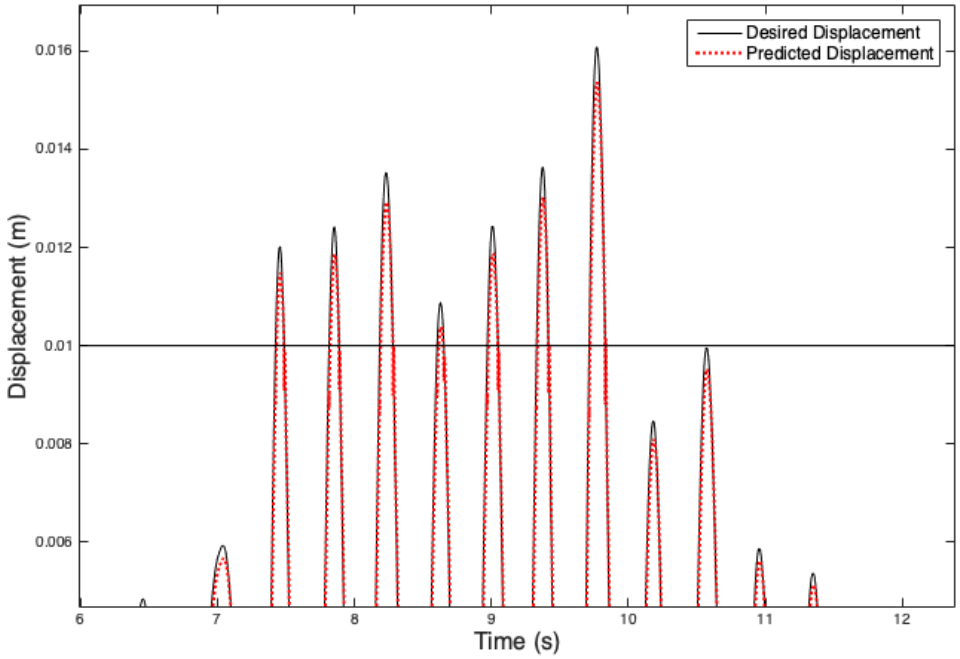


Figure 6.7: A Closer Look at Desired vs. Predicted Displacement

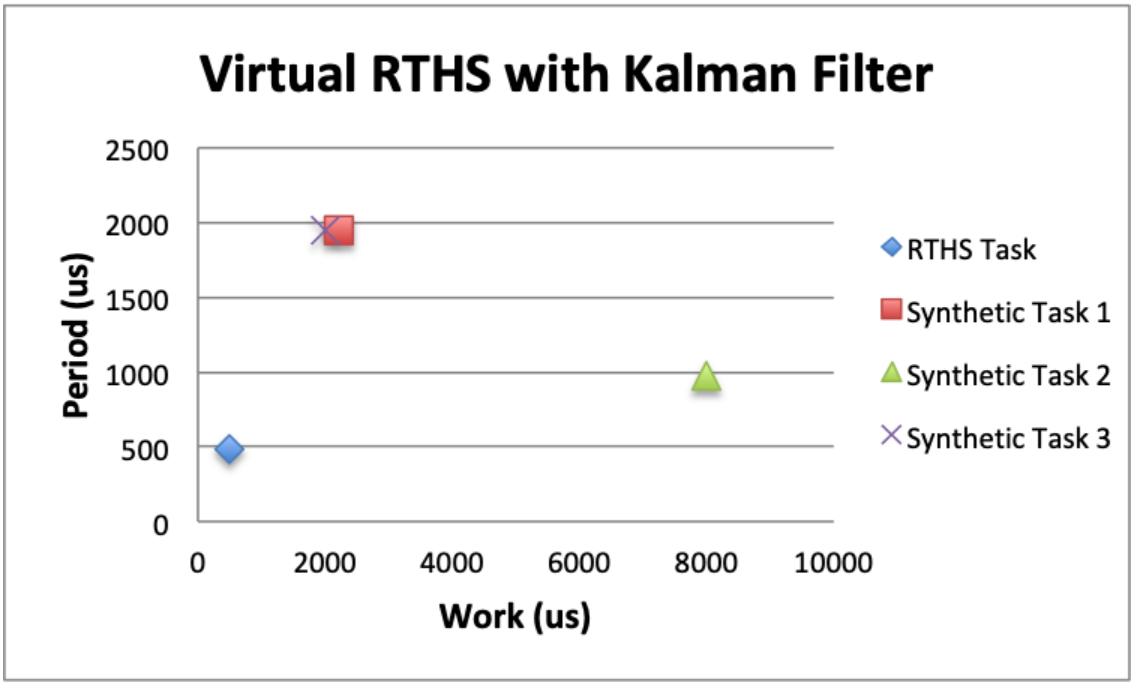


Figure 6.8: System Overview during Kalman Filter Execution

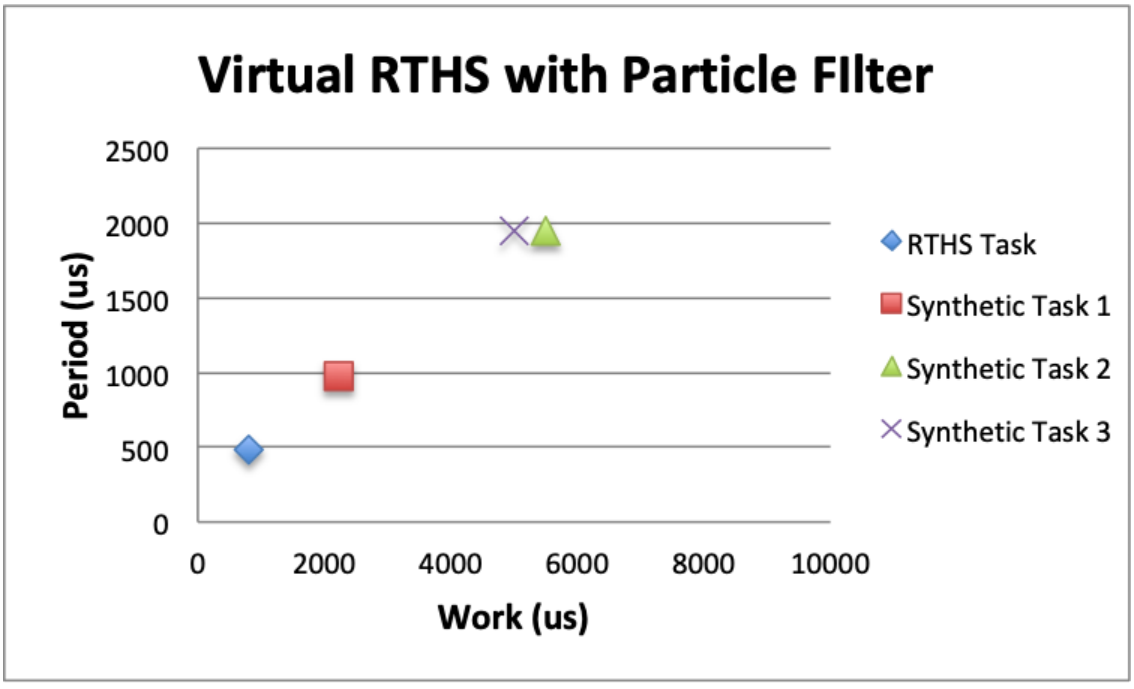


Figure 6.9: System Overview during Particle Filter Execution

As mentioned before, we ran synthetic tasks with the virtual RTHS task that adapted with its mode change, similar to how more elaborate RTHS experiments would do. Figures 6.8 and 6.9 show the workload and period of each task in the system during operation of the Kalman filter and particle filter, respectively. Note that the virtual RTHS task and Synthetic Task 3 adapt their workloads; Synthetic Task 1 adapts its period, and Synthetic Task 2 adapts both its period and workload. Also note that there are only 3 period values used–2048Hz$\approx 488\mu$ sec, 1024Hz$\approx 977\mu$ sec, and 512Hz$\approx 1952\mu$ sec. This is because the estimator must run at a constant 2048Hz and substructure tasks in more complicated RTHS experiments must run at harmonic rates with respect to the main feedback control loop. A normalized root mean squared error (nRMSE) of approximately 0.5% is considered acceptable in the RTHS community. The nRMSE between the estimated and desired displacement shown in Figure 6.6 is 0.267%. Therefore, the virtual RTHS not only successfully transitions modes, but it also performs well.

6.5 Effects of Taskset Discretization

In this section we look at the effect that discretization of tasks' periods and workloads has on schedulability of example task sets through loss of system-wide processor utilization from the continuous version. We begin by randomly generating $10,000$ continuous parallel elastic tasks in the manner described in [42]. Each task is either period-elastic or computationally-elastic, and we schedule these continuous tasks according to the optimal algorithm provided in [42, 43], noting the overall system utilization and objective function value. We then create four discretized task sets from each continuous one by assigning a discretization delta of 0.05, 0.1, 0.2, and 0.5, to each task, meaning we discretize each task in such a way that in the new task sets, there is a candidate utilization every 5%, 10%, 20%, and 50% of the way between $U^{(min)}$ and $U^{(max)}$, plus the endpoints. For example, a period-elastic task with an $T^{(min)} = 0$

and $T^{(max)} = 100$ would be discretized to have candidate period values of 0, 20, 40, 60, 80, and 100 for $\Delta = 0.2$, and it would have candidate period values of 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100 for for $\Delta = 0.1$, etc. We then schedule each of these $40,000$ generated discrete elastic tasks using Algorithm 6, again noting the system utilization and objective function value.

Figures 6.10 through 6.13 show representative results. Figure 6.13 shows the average (and standard deviation of the) system utilization for each level of discretization. Without exception, each discretized task set had a higher (worse) objective function value from Equation 6.2 than the continuous task set from which it was derived. Typically, the objective function value increased with the discretization delta, too, as in the examples shown in Figure 6.10 and Figure 6.12. The single exception in 10,000 task sets is depicted in Figure 6.12. In this case the optimal solution for the task set obtained from $\Delta = 0.1$ occurs when each task selects the utilization value obtained from the 50th percentile. This is exactly the subset of candidate utilizations used to obtain the task set derived from $\Delta = 0.5$ and so also gives the optimal solution for that task set (a subset of the former). However, none of those selected periods are in the task set derived from $\Delta = 0.2$ (a different subset of the former). Therefore, the objective function's value when $\Delta = 0.2$ is necessarily higher. This trend of a (typically) worsening objective function value with an increase of discretization is thus expected. We note that objective function values cannot be compared directly between task sets as they are dependent on tasks' elastic coefficients and maximum utilizations.

For the majority of task sets, system utilization also decreased as a task set became more discretized, as in Figure 6.10. However, because we make scheduling decisions based on the objective function (weighted task utilization) rather than on system utilization, there are cases when making an inferior objective function decision increases task set utilization: this

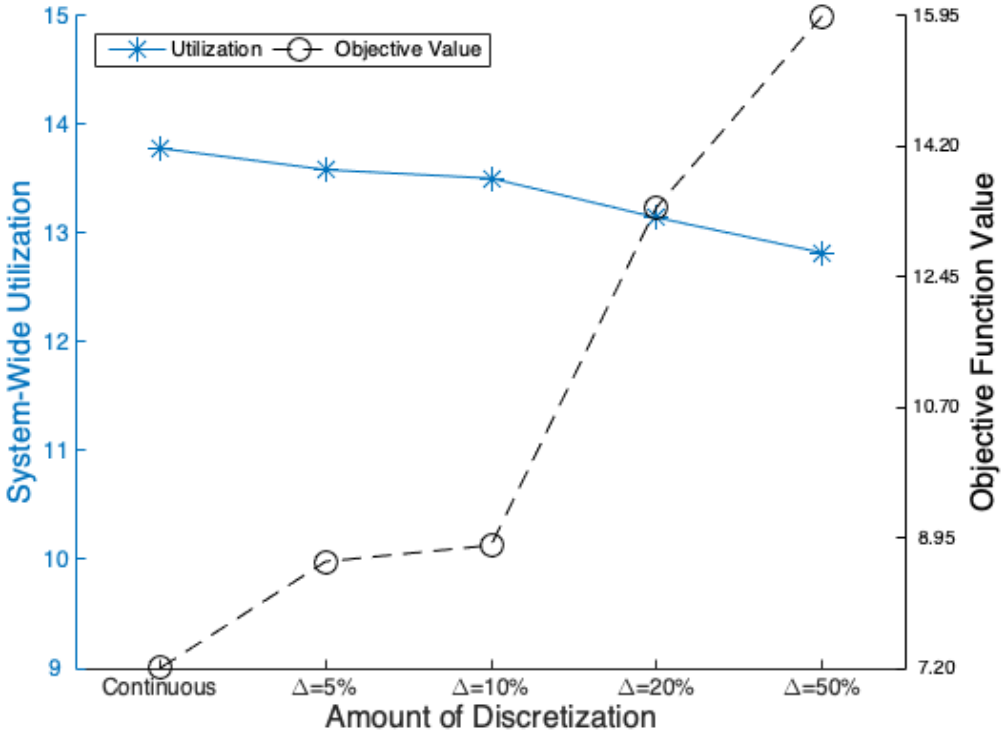


Figure 6.10: Taskset 1 Utilization and Objective Value

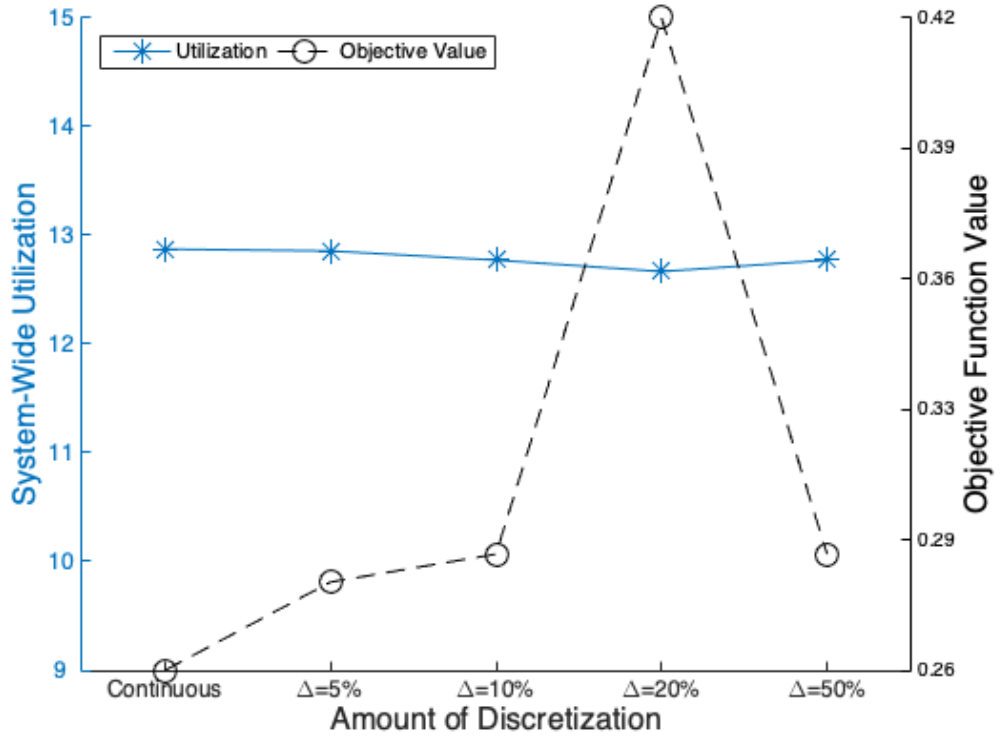


Figure 6.11: Taskset 2 Utilization and Objective Value

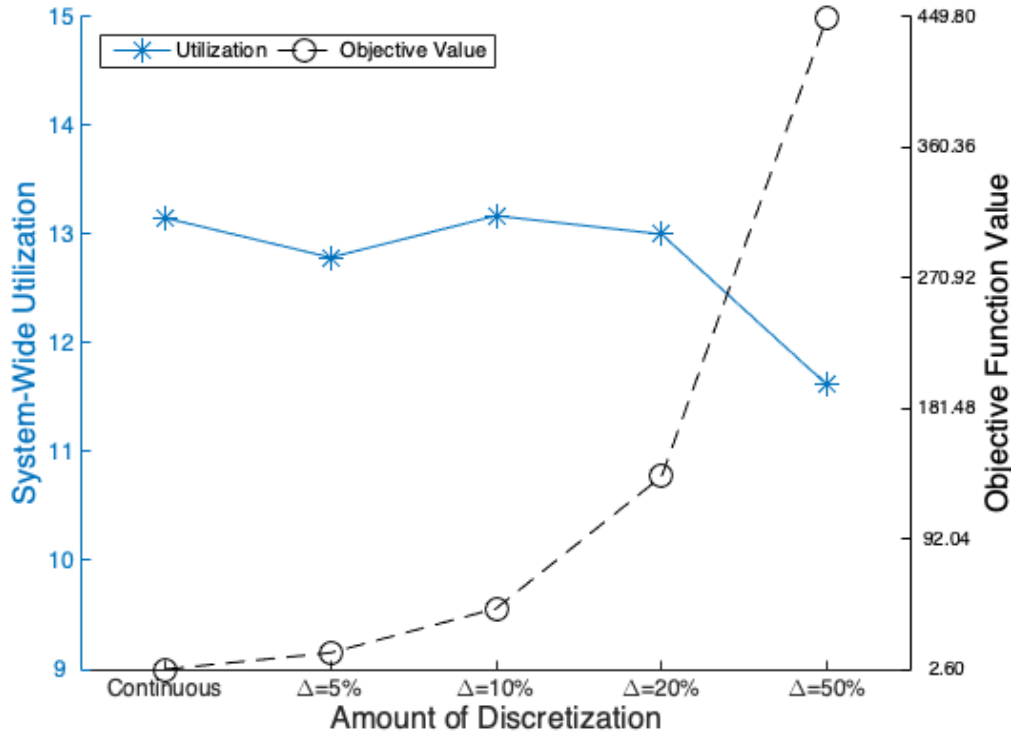


Figure 6.12: Taskset 3 Utilization and Objective Value

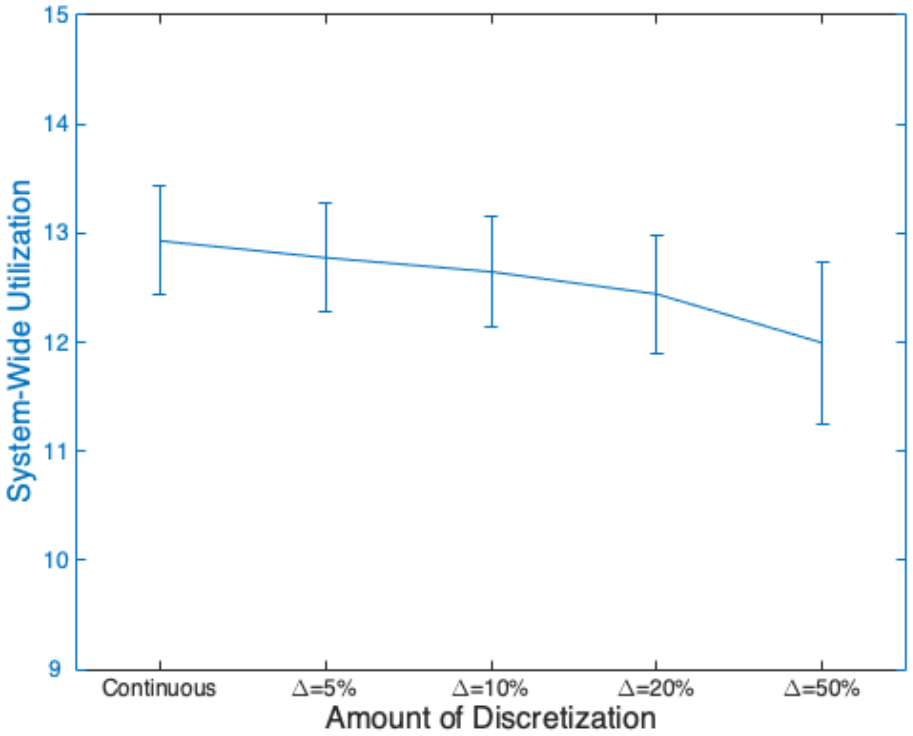


Figure 6.13: Average Utilization (10K Tasksets)

occurred in approximately 18% of task sets (consider Figure 6.12 where $\Delta = 0.1$ gives a higher system utilization than even the continuous version of the task set).

6.6 Summary

In this chapter, we have presented a new elastic task model with discrete sets of possible utilizations for each task. This model allows each task to modify its workload *and/or* its period when changing modes of operation, instead of adapting in only one of those dimensions. This in turn allows a wider range of parallel real-time tasks to exploit elastic scheduling techniques, and also offers a greater diversity of potential adaptations of each task, over a larger region of potential periods and workloads. It is also better aligned with task execution times on realistic hardware.

We have shown how this model can support new real-time hybrid simulations with discretely computationally-elastic, period-elastic, and combined-elastic parallel real-time tasks under the federated scheduling paradigm, via a pseudo-polynomial time scheduling algorithm. We used this scheduling algorithm to implement, for the first time, adaptive resource management to enable adaptive switching between controllers with different computational demands in a virtual real-time hybrid simulation (vRTHS), and examined the effects of scheduling tasks having discretized vs. continuous candidate utilizations in terms of both system utilization and objective function value.

The results presented in this chapter motivate further expansion of this research as future work. Of particular interest is to extend both discrete and continuous elastic scheduling models, and the parallel real-time concurrency platforms that support them, to allow tasks to adapt between low-utilization ($U_i < 1$) and high-utilization ($U_i \geq 1$) modes of operation.

Chapter 7

Conclusion

As real-time systems increasingly utilize both inter-task parallelism and intra-task parallelism, increasingly complex systems may arise. This in turn implies an increased need for adaptive systems that can be modeled as elastic tasks. In this dissertation we have presented several extensions to the elastic task model in order to successfully represent and schedule these tasks on multi-core systems.

Specifically, we expanded the elastic task model from previously only considering sequential tasks on a single processor to now considering both sequential and parallel tasks on multicore systems. We also expanded the concept of task elasticity to not only allow tasks to change their periods (period elasticity), but also to allow some tasks to change their computational loads (computational elasticity), instead. We also allow for sets of tuples with discrete (potentially unique) period and workload combinations rather than merely continuous ranges of acceptable periods (or workloads). This discrete model is perhaps representative of a larger set of real-world adaptive tasks than a continuous model. It allows for combined elasticity in

which tasks can simultaneously adapt their periods and workloads. It also does not force tasks to accept potentially restrictive arbitrary periods or workloads over a range.

In this dissertation we also developed a runtime system for parallel real-time elastic tasks. We then used the system to demonstrate functional equivalence (in terms of scheduling) of continuous period-elastic and computationally-elastic tasks. We also used it to run the first ever adaptive virtual real-time hybrid simulation experiment via the discrete elastic task model. This experiment in turn demonstrated the feasibility of the discrete elastic task model.

7.1 Future Directions

As adaptive real-time systems continue to utilize multiple cores, the work in this dissertation will hopefully lay a foundation for different forms of adaptation. Some preliminary work has already been done by Gill et al. to bring the elastic task model to multi-core mixed-criticality systems [21]. Bringing the algorithms discussed in this dissertation into the operating system kernel or hardware could further increase the types of applications to which this work is relevant by introducing speedups of potentially orders of magnitude. Using other objective functions may be of particular interest in real-time cloud computing where computational demands may vary rapidly and economic factors must be considered.

Elastic scheduling using other parallel scheduling algorithms such as global EDF may be worth exploring. Of particular interest are improvements on federated scheduling, semi-federated scheduling [26] (in which a DAG's structure must be fully known) and reservation-based federated scheduling [49] (which does not require the DAG's structure).

A mechanism that allows for hybrid-elastic tasks to transition during runtime from low-utilization to high-utilization (or vice versa) is still needed. It is also worth exploring whether it is possible to achieve combined utilization with the continuous elastic task model, as are ways to allow task sets consisting of both continuous elastic tasks and discrete elastic tasks, and potentially, to decide among alternative selections of periods and workloads that have equivalent utilizations.

References

[1] P. Antsaklis and J. Baillieul. "Guest Editorial Special Issue on Networked Control Systems." In: *IEEE Transactions on Automatic Control* 49.9 (Sept. 2004), pp. 1421–1423. DOI: 10.1109/TAC.2004.835210.

[2] Sanjoy Baruah. "Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors." In: *IEEE Transactions on Computers* 53.6 (2004).

[3] Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leem Stougie, and Andreas Wiese. "A generalized parallel task model for recurrent real-time processes." In: *Proceedings of the IEEE Real-Time Systems Symposium*. RTSS 2012. San Juan, Puerto Rico, 2012, pp. 63–72.

[4] Sanjoy Baruah, Neil Cohen, Greg Plaxton, and Don Varvel. "Proportionate progress: A Notion of Fairness in Resource Allocation." In: *Algorithmica* 15.6 (June 1996), pp. 600–625.

[5] A. Bastoni, B. Brandenburg, and J. Anderson. "An Empirical Comparison of Global, Partitioned, and Clustered Multiprocessor Real-Time Schedulers." In: *Proceedings of the Real-Time Systems Symposium*. San Diego, CA: IEEE Computer Society Press, 2010, pp. 14–24.

[6] Gregory B Bunting. "Parallel Real-Time Hybrid Simulation of structures using multi-scale models." PhD thesis. Purdue University, 2016.

[7] Giorgio C. Buttazzo, Giuseppe Lipari, and Luca Abeni. "Elastic Task Model for Adaptive Rate Control." In: *IEEE Real-Time Systems Symposium (RTSS)*. 1998.

[8] Giorgio C. Buttazzo, Giuseppe Lipari, Marco Caccamo, and Luca Abeni. "Elastic Scheduling for Flexible Workload Management." In: *IEEE Trans. Comput.* 51.3 (Mar. 2002), pp. 289–302. DOI: 10.1109/12.990127.

[9] Giorgio Buttazzo, Enrico Bini, and Darren Buttle. "Rate-Adaptive Tasks: Model, Analysis, and Design Issuess." In: *Proceedings of DATE 2014: Design, Automation and Test in Europe*. Mar. 2014.

[10] M. Caccamo, G. Buttazzo, and Lui Sha. "Elastic feedback control." In: *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000*. 2000, pp. 121–128. DOI: 10.1109/EMRTS.2000.853999.

[11] John Carpenter, Shelby Funk, Phil Holman, Anand Srinivasan, Jim Anderson, and Sanjoy Baruah. "A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms." In: *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Ed. by Joseph Y.-T Leung. CRC, 2003.

[12] T. Chantem, X. S. Hu, and M. D. Lemmon. "Generalized Elastic Scheduling." In: *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. 2006, pp. 236–245.

[13] T. Chantem, X. Hu, and M. Lemmon. "Generalized Elastic Scheduling for Real-Time Tasks." In: *IEEE Transactions on Computers* 58.4 (Apr. 2009), pp. 480–495. DOI: 10.1109/TC.2008.175.

[14] Thomas Dean and Mark Boddy. "An Analysis of Time-dependent Planning." In: *Proceedings of the Seventh AAAI National Conference on Artificial Intelligence*. AAAI'88. Saint Paul, Minnesota: AAAI Press, 1988, pp. 49–54.

[15] S. K. Dhall and C. L. Liu. "On a Real-Time Scheduling Problem." In: *Operations Research* 26 (1978), pp. 127–140.

[16] Sudarshan Dhall. "Scheduling Periodic Time-Critical Jobs on Single Processor and Multiprocessor Systems." PhD thesis. Department of Computer Science, The University of Illinois at Urbana-Champaign, 1977.

[17] P Emberson, R Stafford, and R.I. Davis. "Techniques For The Synthesis Of Multiprocessor Tasksets." In: *WATERS'10* (Jan. 2010).

[18] 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)*. Oct. 2014, pp. 1–10. DOI: 10.1145/2656045.2656067.

[19] David Ferry, Jing Li, Mahesh Mahadevan, Kunal Agrawal, Christopher Gill, and Chenyang Lu. "A Real-time Scheduling Service for Parallel Tasks." In: *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. RTAS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 261–272. DOI: 10.1109/RTAS.2013.6531098.

[20] David Ferry, Amin Maghareh, Gregory Bunting, Arun Prakash, Kunal Agrawal, Chris Gill, Chenyang Lu, and Shirley 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.

[21] C. Gill, J. Orr, and S. Harris. "Supporting Graceful Degradation through Elasticity in Mixed-Criticality Federated Scheduling." In: *Proceedings of the 6th International Workshop on Mixed Criticality Systems (WMC)*. Dec. 2018.

[22] Joel Goossens, Shelby Funk, and Sanjoy Baruah. "Priority-driven Scheduling of Periodic Task Systems On Multiprocessors." In: *Real Time Systems* 25.2–3 (2003), pp. 187–205.

[23] R. Graham. "Bounds on multiprocessor timing anomalies." In: *SIAM Journal on Applied Mathematics* 17 (1969), pp. 416–429.

[24] J. William Helton. "Orbit structure of the Mobius transformation semigroup action on H-infinity (broadband matching)." In: *Adv. Math. Suppl. Stud.* 1978, pp. 129–198.

[25] W. Horn. "Some simple scheduling algorithms." In: *Naval Research Logistics Quarterly* 21 (1974), pp. 177–185.

[26] 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)*. Dec. 2017, pp. 80–91. DOI: 10.1109/RTSS.2017.00015.

[27] D. S. Johnson. "Near-optimal Bin Packing Algorithms." PhD thesis. Department of Mathematics, Massachusetts Institute of Technology, 1973.

[28] David Johnson. "Fast algorithms for bin packing." In: *Journal of Computer and Systems Science* 8.3 (1974), pp. 272–314.

[29] R. Karp. "Reducibility Among Combinatorial Problems." In: *Complexity of Computer Computations*. Ed. by R. Miller and J. Thatcher. New York: Plenum Press, 1972, pp. 85–103.

[30] Hans Kellerer, Ulrich Pferschy, and David Pisinger. "The Multiple-Choice Knapsack Problem." In: *Knapsack Problems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 317–347. DOI: 10.1007/978-3-540-24777-7{_}11.

[31] 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)*. Apr. 2013, pp. 31–40.

[32] Tei-Wei Kuo and Aloysius K. Mok. "Load Adjustment in Adaptive Real-Time Systems." In: *Proceedings of the IEEE Real-Time Systems Symposium*. 1991, pp. 160–171.

[33] Jaewoo Lee, Kieu-My Phan, Xiaozhe Gu, Jiyeon Lee, A. Easwaran, Insik Shin, and Insup Lee. "MC-Fluid: Fluid Model-Based Mixed-Criticality Scheduling on Multiprocessors." In: *Real-Time Systems Symposium (RTSS), 2014 IEEE*. Dec. 2014, pp. 41–52.

[34] J. Li, K. Agrawal, C. Lu, and C. Gill. "Outstanding Paper Award: Analysis of Global EDF for Parallel Tasks." In: *2013 25th Euromicro Conference on Real-Time Systems*. July 2013, pp. 3–13. DOI: 10.1109/ECRTS.2013.12.

[35] Jing Li, David Ferry, Shaurya Ahuja, Kunal Agrawal, Christopher Gill, and Chenyang Lu. "Mixed-Criticality Federated Scheduling for Parallel Real-Time Tasks." In: *Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Apr. 2016.

[36] Jing Li, Abusayeed Saifullah, Kunal Agrawal, Christopher Gill, and Chenyang Lu. "Analysis Of Federated And Global Scheduling For Parallel Real-Time Tasks." In: *Proceedings of the 2012 26th Euromicro Conference on Real-Time Systems*. ECRTS '14. Madrid (Spain): IEEE Computer Society Press, 2014.

[37] C. Liu and J. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment." In: *Journal of the ACM* 20.1 (1973), pp. 46–61.

[38] J. M. Lopez, J. L. Diaz, and D. F. Garcia. "Utilization Bounds for EDF Scheduling on Real-Time Multiprocessor Systems." In: *Real-Time Systems: The International Journal of Time-Critical Computing* 28.1 (2004), pp. 39–68.

[39] Chenyang Lu, John Stankovic, Tarek Abdelzaher, G. Tao, Sang Son, and M. Marley. "Performance Specifications and Metrics for Adaptive Real-Time Systems." In: *Proceedings of the Real-Time Systems Symposium*. Orlando, FL: IEEE Computer Society Press, Nov. 2000, pp. 13–23.

[40] R McNaughton. "Scheduling with Deadlines and Loss Functions." In: *Management Science* 6 (1959), pp. 1–12.

[41] J. Orr and S. Baruah. "Multiprocessor Scheduling of Elastic Task." In: *Proceedings of the 27th International Conference on Real-Time Networks and Systems, RTNS 2019*. ACM Press, 2019.

[42] J. Orr, C. Gill, K. Agrawal, S. Baruah, C. Cianfarani, P. Ang, and C. Wong. "Elasticity of workloads and periods of parallel real-time tasks." In: *Proceedings of the 26th International Conference on Real-Time Networks and Systems, RTNS 2018*. ACM Press, 2018.

[43] J. Orr, C. Gill, K. Agrawal, J. Li, and S. Baruah. "Elastic scheduling for parallel real-time systems." In: *Leibniz Transactions on Embedded Systems* 6.1 (2019), 5:01–5:14.

[44] Ge Ou, Ali Irmak Ozdagli, Shirley J. Dyke, and Bin Wu. "Robust integrated actuator control: experimental verification and real-time hybrid-simulation implementation." In: *Earthquake Engineering & Structural Dynamics* 44.3 (), pp. 441–460. DOI: 10.1002/eqe.2479. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/eqe.2479.

[45] *Real-Time Scheduling of Parallel Tasks: Theory and Practice*.

[46] Salah Eddine Saidi. "Automatic Parallelization and Scheduling Approches for Co-simulation of Numerical Models on Multi-core Processors." Theses. Université Sorbonne, Apr. 2018.

[47] A Saifullah, K. Agrawal, Chenyang Lu, and C. Gill. "Multi-core Real-Time Scheduling for Generalized Parallel Task Models." In: *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*. Nov. 2011, pp. 217–226.

[48] P Sinha and A. A. Zoltners. "The Multiple Choice Knapsack Problem." In: *Operations Research* 27 (1979), pp. 503–515.

[49] N. Ueter, G. von der Brüggen, J. Chen, J. Li, and K. Agrawal. "Reservation-Based Federated Scheduling for Parallel Real-Time Tasks." In: *2018 IEEE Real-Time Systems Symposium (RTSS)*. Dec. 2018, pp. 482–494. DOI: 10.1109/RTSS.2018.00061.

[50] J. Ullman. "NP-complete scheduling problems." In: *Journal of Computer and System Sciences* 10.3 (1975), pp. 384–393.