USING SIMULTANEOUS MULTITHREADING TO SUPPORT REAL-TIME SCHEDULING

Sims Hill Osborne

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

Chapel Hill
2023

Approved by:

James H. Anderson

Guillem Bernat

Samarjit Chakraborty

Parasara Sridhar Duggirala

Montek Singh

F. Donelson Smith

ABSTRACT

Sims Osborne: Using Simultaneous Mulithreading to Support Real-Time Scheduling
(Under the direction of James H. Anderson)

The goal of real-time scheduling is to find a way to schedule every program in a specified system without unacceptable deadline misses. If doing so on a given hardware platform is not possible, then the question to ask is "What can be changed?"

Simultaneous multithreading (SMT) is a technology that allows a single computer core to execute multiple programs at once, at the cost of increasing the time required to execute individual programs. SMT has been shown to improve performance in many areas of computing, but SMT has seen little application to the real-time domain. Reasons for not using SMT in real-time systems include the difficulty of knowing how much execution time a program will require when SMT is in use, concerns that longer execution times could cause unacceptable deadline misses, and the difficulty of deciding which programs should and should not use SMT to share a core.

This dissertation shows how SMT can be used to support real-time scheduling in both the hard real-time (HRT) case, where deadline misses are never acceptable, and the soft real-time (SRT) case, where deadline misses are undesirable but tolerable. Contributions can be divided into three categories. First, the effects of SMT on execution times are measured and parameters for modeling the effects of SMT are given. Second, scheduling algorithms for the SRT case that take advantage of SMT are given and evaluated. Third, scheduling algorithms for the HRT case are given and evaluated. In both the SRT and HRT cases, using the proposed algorithms do not lead to unacceptable deadline misses and can have effects similar to increasing a platform's core count by a third or more.

For Philip, Joy, and Curtis.

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor, Jim Anderson. He has led me to do better work than I ever thought possible, while at the same time not giving up on me when it inevitably took longer than I ever thought possible. Even if he had not been my advisor, he would still be one of the best professors I have had. Jim has both supremely high standards and endless patience, and I have benefited from both.

I would also like to thank my committee members—Guillem Bernat, Samarjit Chakraborty, Parasara Sridhar Duggirala, Montek Singh, and Don Smith—for their hard work on my behalf. In particular, I thank Don for his direction in timing execution times and also for his excellent class on Operating System Implementation. That class provided key foundations for my work and also, I am convinced, helped me successfully find a job.

Among my fellow students, I have co-written papers with Shareef Ahmed, Joshua Bakita, Jingyuan "Leo" Chan, Saujas Nandi, Nathan Otterness, Stephen Tang, and Tyler Yandrofski. They have all provided valuable contributions. Joshua, my most frequent co-author, deserves special thanks for his knowledge of low-level programming and dedication to answering any and all questions that come his way.

In addition to my co-authors, I have benefited from the expertise and insights of all the members of the real-time group who have been here while I have been—Kechang Yang, Namhoon Kim, Ming Yang, Catherine Nemitz, Tanya Amert, Sergey Voronov, Zelin "Peter" Tong, Clara Hobbs, Shareef Ahmed, Joseph Goh, Syed Ali, Rohan Wagle, and Denver Massey. In particular, Catherine and Tanya have provided encouragement to me during my lowest moments and gave invaluable help to me during my job search.

I could not have come this far without the support of the department's dedicated staff, in particular Murray Anderegg, Robin Brennan, Tatyana Davis, Brandi Day, the late Bill Hayes, Denise Kenney, Jim Mahaney, Mike Stone, Rosario Vila, and Melissa Wood.

I would also like to thank all of the faculty, in particular Ron Alterovitz, Gary Bishop, the late Fred Brooks, Saba Eskandarian, Jasleen Kaur, John Majikes, Dinesh Manocha, Don Porter, and Jan Prins.

I have benefited from the support of my fellow Graduate Women in Computer Science (GWiCS). In addition to the already-mentioned Catherine, Tanya, and Clara, I am particularly grateful for the help and support of Alyssa Barnes, Janine Hoelscher, Bashima Islam, Meghan Stuart, and Kaki Ryan.

Last fall, I had the privilege of teaching an undergraduate class in Algorithms. That class made up my mind to pursue a teaching career, helped me get the job I wanted most, and also improved my own understanding of algorithms. To the Fall 2022 students of COMP 550-02, thank you.

Before coming to Chapel Hill, I was a first an undergrad, and then a Master's Student at UNC-Greensboro. I would not have even considered pursuing a Ph.D. were it not for my advisor there, Francine Blanchet-Sadri. She allowed and encouraged me to set my own research directions and told me "You should really think about getting a Ph.D." When I told her all the reasons I could not get a Ph.D., she encouraged me to find ways around them.

I arrived at UNC-Greensboro having taken only a single introductory course in computer science. The additional faculty and staff who got me from that one class to a Master's Degree and acceptance into UNC-Chapel Hill's Ph.D. program were Mark Armstrong, Richard Cheek, Jing Deng, Lydia Fritz, Lixin Fu, Nancy Green, Fred Sadri, Shan Suthaharan, and Steve Tate in the Department of Computer Science, along with Clifford Smyth and Haimeng Zhang in the Department of Mathematics and Statistics.

The one computer science class I took prior to entering UNC-Greensboro a Chapel Hill summer session of Comp 110, taught by then-Ph.D. student Sean Sanders in the summer of 2014. He introduced me to a subject I have come to love.

I could neither have started nor finished my computer science studies without the support of my parents, Fred and Granville Hill, my in-laws, Lloyd and Rebecca Osborne, and my husband, Philip. I could write an entire second dissertation on everything they have done to support me. It was my parents who first suggested I return to school, and initially paid for me to do so. Lloyd and Rebecca have tirelessly helped care for my children during not only normal working hours, but also endless Saturdays when the workweek had not been enough and when I was out of town or the country for conferences and interviews. Philip has supported me unconditionally in my academic journey by listening to me both brainstorm and complain and by doing far more than his fair share of both housework and childcare. No matter how many times I have had "Just a little more work" turn into multiple late nights and weekends, the only question he has ever asked me is "How can I help?" People have asked me how I can balance having two children with earning a Ph.D. The endless support from my family is the answer.

Finally, I thank my children, Joy and Curtis, for being endless sources of happiness and inspiration and for their patience with their mother having been in graduate school their entire lives.

My work has been supported by NSF grants CNS 1409175, CNS 1563845, CNS 1717589, CNS 215182901, CPS 1837337 CPS 2038855, and CPS 2038960, ARO grants W911NF-17-1-0294, W911NF-20-1-0237, and ONR grant N00014-20-1-2698. In addition, I am grateful to have received the Dissertation Completion Fellowship from the UNC Graduate School.

TABLE OF CONTENTS

LIST OF FIGURES xiv

Chapter 1: Introduction 1

1.1 Simultaneous Multithreading 1

1.2 Real-Time Systems 2

1.3 Hardware Assumptions 3

1.4 Thesis 4

1.5 Contributions 5

1.5.1 Characterizing Execution Times with SMT 5

1.5.2 Soft Real-Time Scheduling 6

1.5.3 Hard Real-Time Scheduling 6

1.5.3.1 Parallelizable Tasks 7

1.6 Organization 8

Chapter 2: Background 10

2.1 Hardware 10

2.2 Real-Time Systems 11

2.2.1 The Sporadic Task Model 11

2.2.2 Correctness Definitions 13

2.3 Real-Time Scheduling Algorithms 14

2.3.1 The Role of Scheduling Algorithms 14

2.3.2 Cyclic Executive Scheduling for HRT Systems 15

2.3.3 Earliest Deadline First Scheduling 17

2.3.3.1 Partitioned EDF 19

2.3.3.2 Global EDF 20
2.3.3.3 Clustered EDF 21
2.3.3.4 Partially Available Cores 23
2.3.3.5 Probabilistic Correctness and Average Costs 25
2.4 Directed Acyclic Graphs 26
2.4.1 Model Overview 27
2.4.2 Scheduling Single Heavy DAGs 28
2.4.3 Federated Scheduling and Related Methods 31
2.5 Timing Analysis 32
2.5.1 Difficulties of Determining WCETs 32
2.5.2 Extreme Value Theory 35
2.5.3 Adding in SMT 37
2.6 Simultaneous Multithreading 37
2.6.1 Technical Information 37
2.6.2 Use Outside of Real-Time Computing 40
2.6.3 Past Uses in Real-Time Computing 40
2.7 Chapter Summary 41
Chapter 3: Timing 42
3.1 Timing Analysis for SRT Systems 42
3.1.1 Scheduling Method Overview 43
3.1.2 Defining an Appropriate Cost 43
3.1.3 Experiments: Setup and Execution without SMT 45
3.1.4 Experiments: Execution Times with SMT 48
3.1.4.1 Summary Statistics 50
3.1.4.2 A More Detailed View 50
3.1.4.3 Key Timing Behaviors Associated with SMT 54
3.1.4.4 The Effects of Varying Inputs 54

3.1.5 Modeling SMT Timing Behavior: SRT 54
3.1.5.1 Overall Expected Value 55
3.1.5.2 Vulnerability 55
3.1.5.3 Interference 57
3.1.5.4 Determining $M_{i(j)}$ 59
3.2 Timing Analysis for HRT Systems: Is SMT Safe? 59
3.2.1 Scheduling Overview and Cost Definition 59
3.2.2 Quantifying Timing Analysis Safety 61
3.2.3 Safety Given a True Random Sample 62
3.2.4 Safety Without True Randomness: Obtaining Execution Times 66
3.2.5 Safety Without True Randomness: Analyzing Execution Times 68
3.2.6 Safety Without True Randomness: Results 69
3.2.7 Rules for Using SMT 70
3.3 HRT Benchmark Results: Characterizing Execution Times 71
3.3.1 The HRT Multithreading Score 71
3.3.2 $M^{(1)}_{i(j)}$ as a Function of $\frac{C_i}{C_j}$ 74
3.3.3 Modeling SMT Timing Behavior: HRT 81
3.4 Conclusions and Future Work 82
Chapter 4: Scheduling Soft Real-Time Systems with SMT 84
4.1 Scheduling With and Without SMT 84
4.2 Dividing the Tasks 93
4.3 Schedulability Studies 97
4.3.1 Schedulability Study Overview 97
4.3.2 Schedulability Test Parameters 98
4.3.3 Schedulability Test Results 99
4.4 Conclusions and future work 104
Chapter 5: Scheduling Hard Real-Time Systems with SMT 105

5.1 CERT-MT: Controlled Execution of Real-Time with Multithreading 105
5.1.1 Scheduling Definition and Limitations 106
5.1.2 Creating a Schedule 108
5.2 CERT-MT Schedulability Studies 112
5.2.1 Schedulability Test Parameters 113
5.2.2 Schedulability Test Results: Four Cores and 60 Second Timeouts 114
5.2.3 Schedulability Test Results: Four Cores and 300 Second Timeouts 118
5.2.4 Schedulability Test Results: Eight Cores 120
5.3 A More Practical Approach to SMT-Aware Scheduling 120
5.3.1 Transforming the System 123
5.3.1.1 Which Tasks Should be Paired? 124
5.3.1.2 Optimization Program Constraints 125
5.3.2 Partitioning the Transformed System 126
5.3.3 Testing Individual Cores 126
5.4 Priority Driven Schedulability Studies 128
5.5 Conclusions and future work 134
Chapter 6: Applying SMT to HRT DAGs 135
6.1 Applying SMT to DAG Tasks 135
6.1.1 Cost and Utilization for DAGs using SMT 135
6.1.2 Optimization Decision Variables and Objective 136
6.1.3 Optimization Constraints 137
6.2 Applying SMT to HRT DAGs: Schedulability Studies 139
6.2.1 Experimental Setup for Single-DAG Systems 139
6.2.2 Single DAG Results 141
6.2.2.1 DAGs With at Most 80 Subtasks 143
6.2.2.2 DAGs with 100 Subtasks 148
6.2.3 Multi-DAG Systems and Federated Scheduling 151

6.2.4 Federated Scheduling Evaluation 152
6.3 Conclusions and Future Work 156
Chapter 7: Conclusion 157
7.1 Summary of Results 157
7.2 Additional Related Work 159
7.3 Future Work 160
Appendix A: Additional Soft Real-Time Graphs 163
A.1 Four Core Graphs 163
A.2 Eight Core Graphs 181
A.3 Sixteen Core Graphs 199
Appendix B: Additional CERT-MT Graphs 217
B.1 Four-Core Graphs, 60-Second Timeout 217
B.2 Four-Core Graphs, 300-Second Timeout 235
B.3 Eight-Core Graphs, 60-Second Timeout 241
B.4 Eight-Core Graphs, 300-Second Timeout 243
Appendix C: Additional Priority-Driven HRT Graphs 245
C.1 Four-Core Graphs 245
C.2 Eight Core Graphs 281
C.3 Sixteen Core Graphs 317
Appendix D: Additional DAG Results 353
D.1 Single DAGs, 10 Subtasks 353
D.2 Single DAGs, 20 Subtasks 389
D.3 Single DAGs, 40 Subtasks 470
D.4 Single DAGs, 80 Subtasks 512
D.5 Single DAGs, 100 Subtasks 536
Appendix E: Additional Federated Scheduling Results 561

E.1 Federated Scheduling, 10 Subtasks 562
E.2 Federated Scheduling, 20 Subtasks 598
E.3 Federated Scheduling, 40 Subtasks 652
BIBLIOGRAPHY 706

LIST OF FIGURES

1.1 Execution without SMT (top) and with SMT (bottom). 2

1.2 A DAG task consisting of six subtasks. 8

1.3 The same DAG task; subtasks v_2 and v_5 paired via SMT. 8

2.1 Cyclic executive scheduling. 16

2.2 Task system scheduled with EDF. 17

2.3 Task system scheduled with NP-EDF. 19

2.4 Example of P-EDF scheduling. 20

2.5 Example of G-EDF scheduling 21

2.6 The Dhall Effect with G-EDF Scheduling. 22

2.7 Cores with limited and full availability. 24

2.8 Scheduling with EDF-HL. 24

2.9 C_i used as a budget within an EDF schedule. 26

2.10 A DAG task consisting of six subtasks. Duplicate of Fig. 1.2 27

2.11 A DAG task scheduled on four cores. 30

2.12 A DAG task scheduled on two cores. 31

2.13 Probability density functions (pdfs) for the Gumbel, Frechet and Weibull distributions with location parameter 0 and scale parameter 1. 36

2.14 Top: task execution without SMT. Middle: execution with SMT. Bottom: execution with SMT, different start times. 38

3.1 Scheduling as if each SMT thread were a separate core. 44

3.2 High-level depiction of measuring costs for an SRT system. 45

3.3 $M_{i(j)}$ values given both C_i and $C_{i(j)}$ are average observed cost. 51

3.4 $M_{i(j)}$ values given both C_i and $C_{i(j)}$ are 99th percentiles 52

3.5 $M_{i(j)}$ values given both C_i and $C_{i(j)}$ are maximum observed values. 52

3.6 Probability density function (pdf, top) and cumulative distribution function (cdf, bottom) for the exponential distributions with expected value $0.25, 0.5$, and 0.75. 56

3.7 Example of simultaneous co-scheduling. 61

3.8 $M^{(1)}_{i(j)}$ values for all pairs. 73

3.9 $M^{(1)}_{i(j)}$ values where $C_j \geq C_i$ holds. 73

3.10 $M^{(1)}_{i(j)}$ values where $C_j \leq C_i$ holds. 74

3.11 $M^{(1)}_{i(j)}$ values where $\frac{C_i}{C_j} \leq 10$ holds. 75

3.12 $M^{(1)}_{i(j)}$ as a function of max $\left(\frac{C_i}{C_j}, 1\right)$ for benchmarks `adpcm_dec`, `adpcm_enc`, `ammunition`,`anagram`, `audiobeam`, and `cjpeg_transupp`. 76

3.13 $M^{(1)}_{i(j)}$ as a function of max $\left(\frac{C_i}{C_j}, 1\right)$ for benchmarks `cjpeg_wrbmp`, `dijkstra`, `epic`,`fmref`, `g723_enc`, and `gsm_dec`. 77

3.14 $M^{(1)}_{i(j)}$ as a function of max $\left(\frac{C_i}{C_j}, 1\right)$ for benchmarks `gsm_enc`, `h264_dec`, `huff_dec`,`huff_enc`, `mpeg2`, and `rijndael_dec`. 78

3.15 $M^{(1)}_{i(j)}$ as a function of max $\left(\frac{C_i}{C_j}, 1\right)$ for benchmarks `rijndael_enc`, `statemate`, and `susan`. 79

4.1 Two configurations of a platform supporting both SMT and non-SMT workloads. 87

4.2 An average scenario for $\mu = 0.6$ 101

4.3 The best scenario for $\mu = 0.6$ 101

4.4 An average scenario for $\mu = 0.4$ 101

4.5 The best scenario for $\mu = 0.4$ 101

4.6 An average scenario for $\mu = 0.2$ 101

4.7 The best scenario for $\mu = 0.2$ 101

4.8 The worst scenario for heavy per-task utilization. 102

4.9 An average scenario for heavy per-task utilization. 102

4.10 The worst scenario for low per-task utilization. 102

4.11 The worst scenario for medium per-task utilization.. 102

4.12 An average scenario for medium per-task utilization. 102

4.13 An average scenario for wide per-task utilization. 102

4.14 An average scenario for the fixed per-task $M_{i(j)}$ distribution. 103

4.15 An average scenario for the exponential per-task $M_{i(j)}$ distribution. 103

4.16 The best 16-core scenario; $h = 0$. 103

4.17 Scenario with $h = 0.25$. 103

5.1 A possible schedule and corresponding x variables for the task system of Ex. 5.2. x(1.1, 2.1, 1, 1)=x(2.1, 1.1, 1, 1)=1 x(1.2, 3.1, 1, 2)=x(3.1, 1.2, 1, 2)=1 x(1.3, 2.2, 1, 3)=x(2.2, 1.3, 1, 3)=1 x(1.4, 3.2, 1, 4)=x(3.2, 1.4, 1, 4)=1 x(4.1, 4.1, 2, 1)=1 x(5.1, 5.1, 2, 1)=0.5 x(4.2, 4.2, 2, 2)=1 x(5.1, 5.1, 2, 2)=0.5 109

5.2 The best scenario. 115

5.3 The worst scenario. 115

5.4 The median scenario. 115

5.5 A mean scenario. 115

5.6 Light per-task utilization. 116

5.7 Moderate per-task utilization. 116

5.8 Wide per-task utilization. 116

5.9 Heavy per-task utilization. 116

5.10 $E[f(1)] = 0.35$, slope 0. 117

5.11 $E[f(1)] = 0.35$, slope 0.3. 117

5.12 $E[f(1)] = 0.55$, slope 0. 117

5.13 $E[f(1)] = 0.55$, slope 0.3. 117

5.14 Light per-task utilization, 300-second timeout. 119

5.15 Moderate per-task utilization, 300-second timeout. 119

5.16 Wide per-task utilization, 300-second timeout. 119

5.17 Heavy per-task utilization, 300-second timeout. 119

5.18 Light per-task utilization, eight cores, 60-second timeout. 121

5.19 Moderate per-task utilization, eight cores, 60-second timeout. 121

5.20 Wide per-task utilization, eight cores, 60-second timeout. 121

5.21 Heavy per-task utilization, eight cores, 60-second timeout. 121

5.22 Light per-task utilization, eight cores, 300-second timeout. 122

5.23 Moderate per-task utilization, eight cores, 300-second timeout. 122

5.24 Wide per-task utilization, eight cores, 300-second timeout. 122
5.25 Heavy per-task utilization, eight cores, 300-second timeout. 122
5.26 Overview of the Priority-Driven approach to SMT scheduling. 124
5.27 The best scenario under all preemption models. 130
5.28 The worst scenario with partial or no preemption. 130
5.29 The worst scenario by RI for full preemption. 130
5.30 The worst scenario by RSA for full preemption. 130
5.31 A median scenario by RI for full preemption. 130
5.32 A median scenario by RI for partial and no preemption. 130
5.33 Light per-task utilization and average SMT parameters. 131
5.34 Medium per-task utilization and average SMT parameters. 131
5.35 Wide per-task utilization and average SMT parameters. 131
5.36 Heavy per-task utilization and average SMT parameters. 131
5.37 Identical parameters to Graph 5.28 apart from core count. 131
5.38 Identical to Graph 5.37 apart from using the exponential distribution. 131
5.39 $E[f(1)] = 0.35$ with slope 0.0. 133
5.40 $E[f(1)] = 0.35$ with slope 0.3. 133
5.41 $E[f(1)] = 0.55$ with slope 0.0. 133
5.42 $E[f(1)] = 0.55$ with slope 0.3. 133

6.1 A DAG task consisting of six subtasks. 136
6.2 The same DAG task; subtasks v_2 and v_5 paired via SMT. 136
6.3 Best 80 subtask scenario. 144
6.4 Best 40 subtask scenario. 144
6.5 Best 20 subtask scenario. 144
6.6 Best 10 subtask scenario. 144
6.7 Example of core count reduced more than utilization 145
6.8 A second example of core count reduced more than utilization 145

6.9 Poor RCC despite good RU. 146
6.10 Erdős-Rényi with $p = 0.5$. 146
6.11 Poor performance using the layer-by-layer method. 146
6.12 Erdős-Rényi with $p = 0.3$. 146
6.13 Layer-by-layer: 2 layers. 147
6.14 Layer-by-layer: 4 layers. 147
6.15 Layer-by-layer: 8 layers. 147
6.16 Erdős-Rényi method. 147
6.17 $K = 40$. 149
6.18 $K = 20$. 149
6.19 $K = 10$. 149
6.20 $K = 1$. 149
6.21 $|V| = 100$, 2 layers, $p = 1.0$, $K = 10$. 150
6.22 $|V| = 100$, 2 layers, $p = 1.0$, $K = 1$. 150
6.23 $|V| = 100$, 2 layers, $p = 0.5$, $K = 10$. 150
6.24 $|V| = 100$, 2 layers, $p = 0.5$, $K = 1$. 150
6.25 Good federated scheduling, light per-dag utilization. 154
6.26 Good federated scheduling, low-heavy per-dag utilization. 154
6.27 Good federated scheduling, medium-heavy per-dag utilization. 154
6.28 Good federated scheduling, high-heavy per-dag utilization. 154
6.29 Poor federated scheduling, light per-dag utilization. 155
6.30 Poor federated scheduling, low-heavy per-dag utilization. 155
6.31 Poor federated scheduling, medium-heavy per-dag utilization. 155
6.32 Poor federated scheduling, high-heavy per-dag utilization. 155

CHAPTER 1: INTRODUCTION

Imagine a very simple drone that must adjust speeds on its rotors every 50 milliseconds. If the drone ever fails to calculate and implement a new set of rotor speeds within this time limit, then it will crash. This drone is a real-time system.

More generally, a real-time system is a computer system that includes programs that must finish on time to be correct. Applications that include real-time components can range from toy drones to airliners to streaming media services. These applications may include dozens of programs, or *tasks*, each with their own deadlines to meet. Typically, each program will need to execute repeatedly, with a separate deadline for each repetition. The goal of real-time scheduling is to find a way to schedule every repetition of every program without unacceptable deadline misses. If acceptably doing so on a given hardware platform is not possible—what counts as acceptable is discussed in Secs. 1.2 and 2.2.2—then the question to ask is "What can be changed?"

It is tempting to say deadline misses can be avoided by using more or better hardware, but doing so is often impractical or even impossible due to limits on hardware size, weight, power, and cost (SWaP-C). For this reason, making the most efficient use possible of a given hardware platform is a vital concern.

One means to do so, the focus of this dissertation, is simultaneous multithreading.

1.1 Simultaneous Multithreading

Simultaneous multithreading (SMT) is a technology that allows a single computer core to execute multiple programs at once, at the cost of increasing the time required to execute individual programs (Eggers et al., 1997; Tullsen et al., 1995).

Example 1.1. At the top of Fig. 1.1, two programs execute sequentially without SMT. The dark-colored program begins at time 0 and finishes at time 6.0. The light-colored program begins at time 6.0 and finishes at time 12.0.

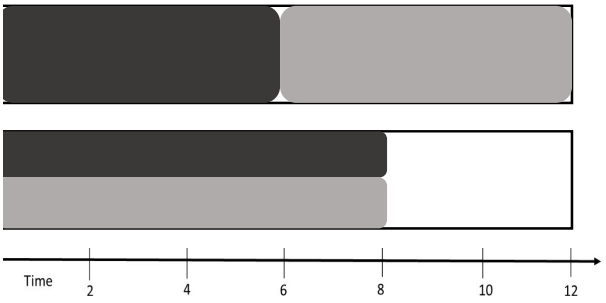

Figure 1.1: Execution without SMT (top) and with SMT (bottom).

In the bottom of the figure, the same programs use SMT to execute in parallel on a single core. They execute more slowly than before, but both programs now finish at time 8.0: earlier for the light program, but later for the dark program. ◊

In the example, SMT allows two programs, rather than one, to finish within 8.0 time units. The potential benefits of completing more work within a set amount of time are clear. Unfortunately, SMT poses two problems for real-time work. First, SMT increases the execution time of individual programs. Second, SMT causes every program's execution time to be dependent on what other programs are executing at the same time, potentially making execution times less predictable. For these reasons, SMT has been largely ignored by the real-time community, despite its widespread adaptation in other areas.

This dissertation demonstrates the value of using SMT within a real-time context. Specifically, we show how to overcome the problems SMT poses for real-time systems and how to use SMT to increase the number of systems that can be scheduled on a given hardware platform without unacceptable deadline misses. We consider both hard real-time (HRT) systems, where no deadline misses are acceptable, and soft real-time (SRT) systems, where limited deadline misses are acceptable. For the HRT case we consider both the case where individual programs are limited to sequential execution and also the case where each program is modeled as a directed acyclic graph (DAG) consisting of may subtasks that may be executed in parallel.

1.2 Real-Time Systems

A real-time system τ is defined as a number of *tasks*, denoted τ_1, τ_2,...τ_n. A task is a program that will be repeatedly invoked, such as flight stabilization in our drone model. Each individual invocation is termed a *job*. We give a more detailed description of real-time systems in Sec. 2.2.

Tasks are defined, in part, by their deadlines. For HRT systems, τ is correctly scheduled if and only if all jobs finish before their deadlines. As real-time systems are often assumed to continue running forever, systems can include an infinite number of jobs. For SRT systems, τ is correctly scheduled if and only if the maximum amount of time by which any job will miss its deadline has, with high probability, a finite upper bound. We elaborate on this definition in Sec. 2.2.2.

Two key characteristics of any real-time system are its *feasibility* and *schedulability*.

Definition 1.1. A system is *feasible* on a given hardware platform if it is possible to correctly schedule it on that platform. ◄

Definition 1.2. A system is *schedulable* under scheduling algorithm A on a given hardware platform if using algorithm A to decide when each job executes results in the system being scheduled correctly. ◄

We are interested in schedulability under practical algorithms. Informally, we say a scheduling algorithm is practical if its real-world performance is close to its theoretical performance,which relies on simplifying assumptions. We do not attempt to formally define what makes an algorithm practical, but instead will discuss the practicality of existing scheduling algorithms in Sec. 2.3 and of algorithms we propose as they come up in Chapters 4 and 5.

1.3 Hardware Assumptions

The ability to correctly schedule a real-time system is dependent on the hardware platform used. We consider platforms of m identical CPU cores for scheduling. The practical result of identical CPU cores is that a task's execution time will be constant regardless of where it is executed. This model is simpler than many actual hardware platforms. In particular, it does not account for different computing cores having separate caches. In practice, preempting a task and then resuming its execution on a different core may lead to increases in execution time due to cache affinity loss, an issue we discuss further in Secs. 2.1 and 2.3.

SMT may be enabled on a per-core basis. If SMT is in use, then it provides two hardware threads per core. These threads are identical; given that two programs are executing on a single core, it does not matter which program is assigned to which hardware thread. When two tasks are sharing a core, our analysis assumes it is not possible to speed up one task at the expense of the other. Our assumptions regarding SMT are discussed in more detail in Sec. 2.6

1.4 Thesis

We have found that careful use of SMT can increase the number of real-time systems that are schedulable on a given hardware platform. Equivalently, using SMT can decrease the number of computer cores needed to correctly schedule a given system. This result holds for both HRT and SRT systems. Our thesis builds on our findings.

Thesis: *For a wide variety of both soft and hard real-time systems, simultaneous multithreading (SMT) can be used to make otherwise infeasible systems schedulable by practical algorithms. The results of using SMT can be similar to the results of increasing the number of available computer cores by 25% or more. Using SMT does not weaken existing real-time guarantees. The effects of using SMT can be seen both analytically and empirically.*

We consider SRT and HRT separately. Within each of these areas, we show how to determine execution times with SMT and how to apply SMT to a system to get the maximum use out of a given hardware platform.

A key part of our thesis is that we do not weaken existing real-time guarantees. This claim requires some background on real-time scheduling without SMT. The strongest possible guarantee of correctness would be knowing all tasks' *worst-case execution times* (WCETs) with perfect confidence *and* having a mathematical proof that given those WCETs, no deadline will ever be missed.

Unfortunately, perfect knowledge of WCETs is impossible, in part because with SMT, a task's execution time requirement is dependent on the scheduling of other tasks. However, this problem is not exclusive to systems using SMT: knowing execution times with absolute certainty is essentially impossible on any multicore platform (Wilhelm, 2020). Even so, the relevance of multicore scheduling research is unquestioned.

There are two ways to reconcile the relevance of multicore scheduling research with the weakness of any guarantee based on potentially incorrect WCETs. First, for some applications the possibility of deadline misses due to incorrectly stated WCETs may be considered an acceptable risk. In this case, showing that SMT does not increase the probability of stated WCETs being incorrect implies that the possibility of deadline misses due to incorrectly stated WCETs in the presence of SMT is also an acceptable risk.

Second, avoiding deadline misses due to scheduling tasks with known execution times and avoiding deadline misses due to unknown execution times are separate concerns. This view allows results in multicore scheduling to be significant while understanding that many applications must wait on a breakthrough in determining execution times or a significant change in multicore architecture.

Both of these arguments apply to SMT scheduling as well as to multicore scheduling. Our burden is not to show that execution times with SMT can be perfectly determined, but rather to demonstrate that SMT does not cause real-time guarantees to be any weaker than they presently are in multicore scheduling.

1.5 Contributions

In this section, we describe the contributions that support our thesis: we measure how SMT affects execution times in practice and develop SMT-aware methods for scheduling both SRT and HRT systems.

1.5.1 Characterizing Execution Times with SMT

The ability of SMT to support real-time scheduling is dependent on both how SMT affects execution times and how confident we can be in our knowledge of those execution times. For this reason, our first contribution is to measure how SMT affects execution times and to assess our confidence in those measurements.

With SMT, a task's execution time will be influenced by any task being executed on the other half of the same core. We therefore measure each task's execution time for every possible task with which it could share a core. For HRT tasks, we reduce execution-time variation by requiring that two jobs sharing a core must begin execution simultaneously.

For HRT tasks, we are estimating WCETs. For SRT tasks, knowing *average-case execution times* (ACETs) is often sufficient to determine if a system is correctly scheduled. For this reason, upper-bounding ACETs of tasks using SMT may be acceptable for SRT systems.

We empirically assess our estimates via additional observations. This process serves two purposes. First, by showing that execution times can be reliably estimated, we show that meaningful comparisons of schedulability with and without SMT are possible and we support our claim that SMT need not weaken real-time guarantees. Second, characterizing SMT's effects on execution allows us to generate an infinite number of synthetic real-time systems. Synthetic task systems are a key method for evaluating the effectiveness of different scheduling algorithms both in this dissertation and in real-time research generally. However, such evaluations are meaningful only if generated systems exhibit plausible behaviors. For example, we could easily show that SMT is always beneficial in cases where SMT does not increase execution times at all, but this case is essentially non-existent, making such a result meaningless. By analyzing the effects of SMT on

real benchmark programs, we can use the results of experiments conducted on synthetic systems to provide meaningful insights about the real world.

1.5.2 Soft Real-Time Scheduling

Our next contribution, presented in Chapter 3, is to show the benefits of SMT in scheduling SRT systems.

As knowing ACETs is often sufficient for SRT scheduling, it follows that much of the execution time cost uncertainty caused by SMT is tolerable in an SRT settting. For this reason, tasks using SMT are scheduled using existing scheduling methods as if each hardware thread were a separate core; at any time, any task using SMT can execute alongside any other task using SMT.

However, not all tasks should use SMT; in some cases, the increase in execution time caused by SMT will be too great. For example, if SMT would more than double a task's execution time, then using SMT for that task would make scheduling the systems harder. Therefore, SRT tasks should be divided into at least two subsystems: one that does use SMT, and one that does not. Each subsystem can then be scheduled on a dedicated set of processors that have SMT enabled or disabled as appropriate.

We give a mathematical test for determining whether a given workload can be correctly scheduled on a platform with SMT and empirically evaluate the benefits of SMT for SRT systems.

1.5.3 Hard Real-Time Scheduling

Our next contribution is to show how SMT can be used to schedule HRT systems. Since the requirement in these cases is to never miss a deadline, execution times with SMT must be known to the same level of certainty as execution times without SMT. For this reason, our HRT work focuses on scheduling pairs of tasks whose joint execution times have been determined, as discussed in Section 1.5.1.

This approach gives us two problems to consider: which tasks should be paired and how those pairs should be scheduled. As pairing tasks influences execution times, which in turn influence the range of possible scheduling choices, finding an optimal solution—i.e. the minimum number of cores necessary to schedule a given HRT system under the paired-jobs paradigm—would have exponential complexity.

We do give an optimal algorithm that is guaranteed, given sufficient time, to schedule any feasible system, but find that it is practical only for small task systems. For larger systems, we consider two heuristic approaches: first pairing tasks heuristically and then scheduling the resulting pairs, or first dividing a task system into smaller subsystems and then scheduling the smaller subsystem either optimally or through use of

a task-pairing heuristic. We find that through these heuristics, it is possible to obtain results similar to those that use the optimal approach.

We consider one variation on HRT scheduling that uses different assumptions.

1.5.3.1 Parallelizable Tasks

In the simplest model of a real-time system, jobs are not parallelizable. SMT may allow for the execution of more jobs at once, but its parallelism cannot decrease the time needed to execute indiviudual jobs.

However, parallelism is an essential part of scheduling real-time workloads in modern applications such as image recognition (Amert et al., 2021; Elliott et al., 2015; Houssam-Eddine et al., 2019; Yang et al., 2015), autonomous vehicles (Guo et al., 2019; Houssam-Eddine et al., 2021; Kato et al., 2015), and aviation (Melani et al., 2017). When the total processor time required for a task exceeds the task's relative deadline—a frequent occurrence in some domains—scheduling a task without parallelism becomes impossible.

Such tasks can be completed in a timely manner if they include segments that can be executed in parallel. These parallelizable workloads can be modeled using directed acyclic graphs, or DAGs. A DAG task is modeled in part as a graph $G = (V, E)$ with V denoting a set of vertices and E denoting a set of edges. Each vertex $v \in V$ represents a portion of the task, or *subtask*, and each directed edge (v_i, v_j) indicates that subtask v_i must be completed before subtask v_j can begin. Vertices that are not connected may be executed in parallel (Baruah et al., 2012). SMT can be used to improve the schedulability of DAG tasks by executing selected vertices in parallel on a single core.

Example 1.2. Consider the DAG shown in Fig. 1.2. Each subtask has its own execution cost, giving the overall DAG-task a total execution cost of 130. The DAG's length gives the minimum amount of time needed for it to complete given unlimited processors. In this case, the DAG can complete in 70 time units given two processors by allowing v_2 and v_4 to execute on one core in parallel to v_3 and v_5 on a second core.

Suppose that if subtasks v_2 and v_5 use SMT to execute on a single core, then 75 time units are required for both to complete. Then applying SMT to v_2 and v_5 produces the DAG shown in Fig. 1.3. All precedence constraints from the original DAG are maintained. The new DAG has a reduced total execution cost of 105 time units. ◊

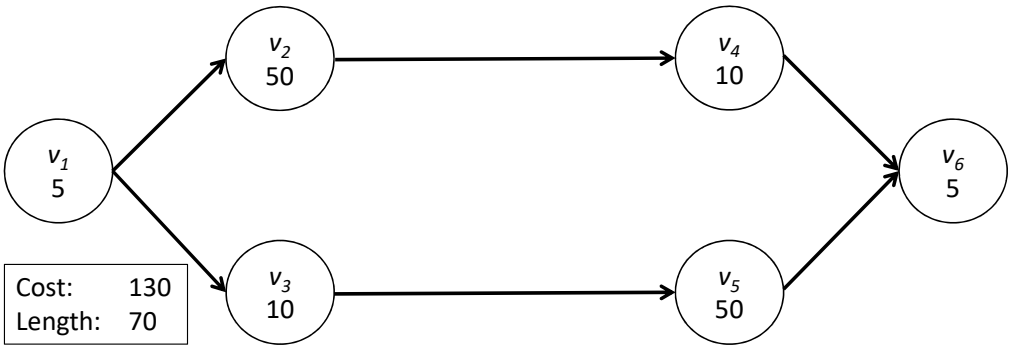

Figure 1.2: A DAG task consisting of six subtasks.

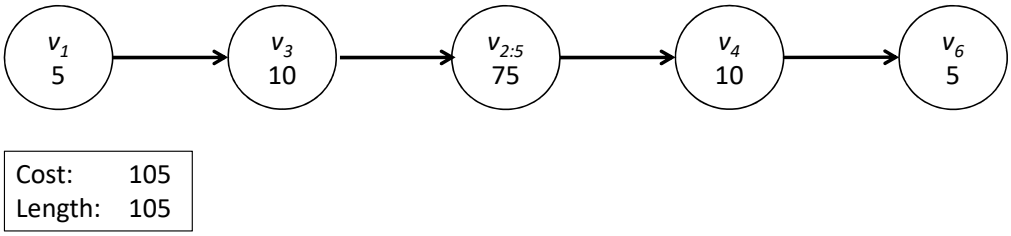

Figure 1.3: The same DAG task; subtasks v_2 and v_5 paired via SMT.

Selecting vertices with which to use SMT is complicated by the presence of precedence constraints. Clearly, two subtasks with a precedence relationship cannot be scheduled at the same time. More subtly, scheduling subtasks to execute together can create additional precedence constraints.

Example 1.3. In Fig. 1.2, executing v_5 and v_4 in parallel with SMT would effectively make both of v_2 and v_3 predecessors to both v_4 and v_5, potentially eliminating the benefits of using SMT. ◊

We give an algorithm that applies SMT to a DAG task so as to minimize its utilization without introducing additional precedence constraints that make the DAG unschedulable. By doing so, we can make dramatic reductions in both the utilization and core requirements of DAGs. Our algorithm is optimal in terms of minimizing total utilization, but has exponential time complexity. To assist with using SMT with large DAGs, we include a tunable parameter that allows for tradeoffs between optimality and running time. We evaluate our work in terms of both individual DAGs and of systems of DAG tasks.

1.6 Organization

The rest of this dissertation is organized as follows. In Chapter 2, we present background information. In Chapter 3, we present our findings on the effects of SMT on execution times and our confidence in our

execution time estimates. We give our work on SRT scheduling in Chapter 4. We give our work on the basic version of HRT scheduling in Chapter 5, and on our DAG scheduling variation in Chapter 6. Finally, in Chapter 7 we summarize our findings and suggest directions for future work. Additional materials, including the code used to produce our results, are available online at `https://www.cs.unc.edu/~shosborn/dissertation`.

CHAPTER 2: BACKGROUND

In this chapter, we present the existing work that this dissertation builds on. Specifically, we provide more detailed information on our assumed hardware platform, the modeling and scheduling of real-time systems, the challenges of determining execution times on multicore platforms, and SMT.

2.1 Hardware

In this section, we provide more detail on the hardware model briefly described in Section 1.3. We assume the use of a Symmetric Multiprocessor (SMP). In this design, each individual core has one or more levels of cache available only to that core. In addition, *clusters* of cores share a common *lowest-level cache* (LLC). All cores have equal access to shared DRAM memory. Typical access times are in the range of 0.5-1.0 ns for L1 cache, 3-10 ns for L2 cache, 10-20 ns for L3 cache (typically the lowest level), and 50-100 ns for main memory (Hennessy and Patterson, 2019).

If a process is preempted, its total execution time may be increased due to *cache affinity loss.* If a preempted process *migrates*, or resumes execution on a different core, bringing needed data into the new core's cache will take time, increasing total execution cost. This cost will typically be greater if execution resumes on a different cluster, as the process will have lost access to all cache, not merely the higher levels of cache. Even when a process resumes execution on the same core, cache affinity loss will occur if other processes evict the original process' data from cache. However, eviction can often, though not always, be avoided by using cache partitioning to assign each process a dedicated fraction of the cache (Altmeyer et al., 2014, 2016; Bakita et al., 2021; Chisholm et al., 2015, 2017; Guo et al., 2020; Kim et al., 2013; Kirk, 1988; Liedtke et al., 1997; Mueller, 1995; Wolfe, 1993; Ye et al., 2014). We do not explicitly consider the effects of cache affinity loss in our work, but our choice of scheduling algorithms, discussed in Section 2.3, is informed by existing works that do so.

2.2 Real-Time Systems

In this section, we detail how real-time systems are modeled and scheduled in existing work. Some of the information here duplicates that in Section 1.2, but is included to make reading easier.

2.2.1 The Sporadic Task Model

We use the *sporadic task model*. In this model, real-time systems are modeled on the basis of *tasks* and *jobs*. A task is a recurring computation and a job is an instance of a task tied to a particular time or input set (Liu and Layland, 1973). A task system—denoted τ—consists of n tasks, denoted $\tau_1,\ \tau_2, \ldots \tau_n$. Each task is defined by its *cost*, denoted C_i, and *period*, denoted T_i (both terms are discussed below). We write $\tau_i = (C_i, T_i)$.

We assume that time is discrete but not necessarily integral. When needed, we use ε to denote the smallest possible unit of time; all costs and periods are integer multiples of ε. This is an unusual view of time; most works use either continuous time or time that is both discrete and integral. In our case, expressing the effects of SMT on execution time is easier when we do not restrict ourselves to integer time, but continuous time would make our analysis of DAGs more difficult.

Each instance of a task is a *job*. When a job becomes available for execution, it is *released* by the associated task. The maximum difference between the completion and release times of any job of τ_i is given by τ_i's *response time*, denoted R_i. Unless otherwise specified, all jobs and tasks are strictly sequential; jobs cannot execute in parallel either with themselves or with other jobs of the same task. If the underlying task system is *periodic*, then every task τ_i releases a job once every T_i time units (Liu and Layland, 1973). If the system is *sporadic*, then every task τ_i releases a job *at most* once every T_i time units (Mok, 1983). We assume *implicit* deadlines: all jobs have *absolute deadlines* equal to their release times plus T_i.

The exact definition of task cost can vary and will be specified for each problem we consider. The simplest and most intuitive definition for C_i is that it gives τ_i's WCET. This definition is analytically convenient, but can be problematic given the difficulties of determining WCETs that will be discussed in Section 2.5 below. Alternatively, C_i can be defined as a task's ACET or even as a particular percentile; for example, we could define a task so that C_i will be greater than the actual execution time for 99% of all jobs.

If C_i is less than τ_i's WCET, then a scheduling algorithm must include provisions for the actual execution time of some job $\tau_{i,j}$ exceeding C_i. We discuss this possibility in Section 2.3.3.5. When not stated otherwise, we assume that C_i gives a task's WCET.

The *utilization* of task τ_i—intuitively, the fraction of a core that the task needs in the long run—is given by $u_i = \frac{C_i}{T_i}$. The total utilization for the system is given by

$$U = \sum_{i=1}^{n} u_i. \tag{2.1}$$

We assume that all tasks are independent; no precedence constraints exist between jobs of different tasks. Furthermore, we assume there are no critical sections or other constraints that could limit specific task combinations from executing concurrently. Given these assumptions, a task system is feasible on m cores if and only if

$$\forall i : u_i \leq 1 \text{ and} \tag{2.2}$$

$$U \leq m \tag{2.3}$$

hold. This result applies to both sporadic and periodic systems.

For HRT systems, the necessity of (2.2) is self evident: $u_i > 1$ implies $C_i > T_i$, which in turn implies no deadline for any job of τ_i will be met. Exp. (2.2) is necessary for SRT systems as well: should $C_i > T_i$ hold, then the first job of τ_i will have a minimum tardiness of $C_i - T_i$ time units. Assuming all jobs release as fast as possible, the minimum tardiness will increase by $C_i - T_i$ with every subsequent job of τ_i, leading to unbounded tardiness.

As for (2.3), its necessity was given by Leung and Merril for periodic systems (Leung and Merrill, 1980). As periodic systems are a subset of sporadic systems, this result implies necessity for sporadic systems as well. The sufficiency of both conditions for sporadic systems was proved by Mok (Mok, 1983).

Note that not all feasible task systems can be correctly scheduled in practice. An algorithm capable of correctly scheduling all feasible systems is *optimal.*

Definition 2.1. A scheduling algorithm is *optimal* if and only if it will correctly schedule all feasible task systems. ◄

Particularly for HRT systems, optimal scheduling algorithms capable of scheduling all feasible systems tend to rely on unrealistic assumptions such as zero-cost context switches. For example, PFair scheduling requires subdividing each task into many subtasks so that the task will be preempted at each subtask boundary (Baruah et al., 1993; Srinivasan and Anderson, 2006). When the time taken up by these preemptions is accounted for, PFair algorithms are inferior to many scheduling algorithms that are not theoretically optimal (Brandenburg, 2011).

2.2.2 Correctness Definitions

An HRT system is correctly scheduled if it can be shown that so long as cost values C_i are accurate for all tasks, no deadline will ever be missed. Formally,

Definition 2.2. An HRT system is correctly scheduled if and only if the following holds:

$$\forall i : R_i \leq T_i. \blacktriangleleft \tag{2.4}$$

For SRT systems, a job's *tardiness* is the difference between its completion time and deadline, if the job completes after its deadline, and zero otherwise. A task's tardiness is the maximum tardiness of any of its jobs. In our model, the scheduling of an SRT system is *strictly correct* if *bounded tardiness* is guaranteed, i.e. a finite upper limit on tardiness exists for all tasks. Formally,

Definition 2.3. An SRT system is strictly correct if and only if it is scheduled such that the following holds:

$$\forall i : R_i < \infty. \blacktriangleleft \tag{2.5}$$

What we term strictly correct is the only correctness condition considered in much of the literature considering bounded tardiness. Achieving strict correctness requires that WCETs are known or upper-bounded for all tasks.

As an alternative to strict correctness, we say that scheduling for an SRT system is *probabilistically correct* if the bounded tardiness property holds with probability approaching one. We state this requirement more formally in terms of *probabilistic deadlines*:

Definition 2.4. τ_i meets the *probabilistic deadline* of (δ_i, η_i) if the probability of any job's response time exceeding δ_i is at most η_i. $\blacktriangleleft$

Definition 2.5. τ is *probabilistically correct* if all tasks meet probabilistic deadlines for which $\delta_i < \infty$ and η_i approaches one. ◄

In our review of scheduling algorithms below, we discuss conditions for strict SRT correctness first in Section 2.3 and then discuss the modifications needed to achieve probabilistic correctness in the presence of unknown WCETs in Section 2.3.3.5. We consider probabilistic correctness to be a variation of the bounded tardiness approach.

Other definitions for SRT correctness exist. The *weakly hard* model was introduced by Bernat et al. in 2001 for systems that can tolerate limited deadline misses. In that model, a system is correct so long as at most h deadlines are missed by any sequence of K consecutive jobs of a given task (Bernat et al., 2001). [1]

This model has received considerable attention in recent years (Choi et al., 2021; Hammadeh et al., 2019; Huang et al., 2019; Pazzaglia et al., 2020; Sun and Natale, 2017), motivated largely by new use cases in embedded real-time systems and the Internet of Things.

Predating both the bounded tardiness and weakly-hard correctness definitions, some sources define SRT correctness in terms of the percent of deadline misses occurring over an observed interval. One such example is Jain et al.'s (Jain et al., 2002) paper comparing SRT scheduling with and without SMT, which is discussed in more detail in Section 2.6. This definition has since been largely replaced by correctness definitions such as bounded tardiness and weakly hard that consider the behavior of a theoretically infinite stream of jobs.

2.3 Real-Time Scheduling Algorithms

In this section, we discuss the role of scheduling algorithms in general, followed by an overview of common scheduling algorithms used for both HRT and SRT systems. When not otherwise specified, we continue to assume that for all tasks, C_i is at least the task's WCET.

2.3.1 The Role of Scheduling Algorithms

A task system is executed on a specified platform using a scheduling algorithm responsible for determining when each job will execute and on what core. In our model, real-time systems are assumed to keep executing forever. Consequently, correctness requires either a pre-computed job-by-job schedule that can be

[1] The original paper uses different notation; (h, K) is the most common notation for this model used today.

repeated indefinitely, or an algorithm that will provably schedule all jobs to complete by their deadlines (for HRT) or within a bounded margin of their deadlines (to achieve strict correctness for SRT).

Of the many real-time scheduling algorithms that are the subject of ongoing research, we focus on two families of scheduling algorithms: *Cyclic Executive* (CE) scheduling and *Earliest Deadline First* (EDF) scheduling. Our goal in this section is to justify our use of these algorithms in our work and provide sufficient context on them for our work to be understood.

2.3.2 Cyclic Executive Scheduling for HRT Systems

CE scheduling was developed to schedule HRT, periodic systems with high predictability and low run-time overheads (Baker and Shaw, 1989). Disadvantages to CE scheduling include potentially high costs to compute a schedule, the necessity of computing an entirely new schedule should even one task in the system need to be changed, and, in many cases, frequent preemptions.

The essence of CE scheduling is to pre-compute a table stating when every job should execute. A correct schedule is built by assigning jobs to equal-length intervals called *frames*—one job can span multiple frames—such that no job is assigned to a frame that begins before its release and all jobs will complete in a frame that ends no later than each job's deadline. It follows that the maximum frame length is given by the shortest period in the system. Multiple jobs can be assigned to a single frame so long as the total scheduled execution time does not exceed the length of the frame. A key concept for CE schedules is the *hyperperiod.*

Definition 2.6. The longest common multiple of all task periods in a system gives the system's *hyperperiod*, denoted H. If all periods apart from the smallest are integer multiples of all smaller periods, then the system is *harmonic.* ◄

A CE schedule is correct if the rules stated in Def. 2.7 below, adapted from Baker and Shaw (Baker and Shaw, 1989), hold.

Definition 2.7. A CE schedule is *correct* if over the course of each hyperperiod: (i) all jobs are scheduled; (ii) any non-preemptable job is scheduled in one frame; (iii) every job completes in a frame that ends no later than its deadline; (iv) no job executes in a frame that begins before its release; (v) the total execution time scheduled in each frame is no greater than the frame size; and (vi) no job executes in parallel with itself. ◄

For unicore systems, determining a correct schedule for a given frame size, or else determining that no such schedule exists, can be computed with time complexity that is polynomially related to the number of

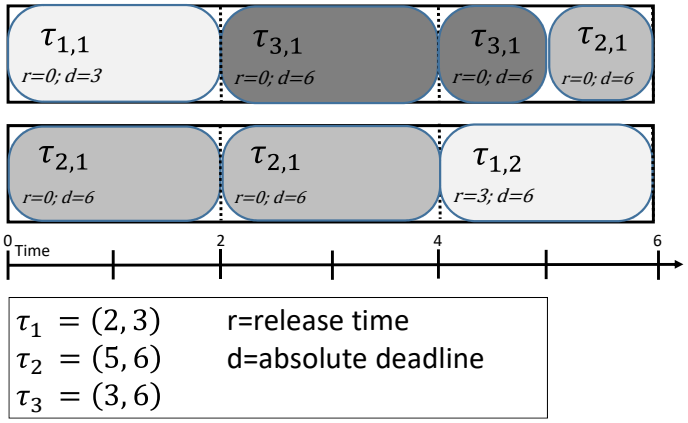

Figure 2.1: Cyclic executive scheduling.

jobs in the hyperperiod plus the number of frames in the hyperperiod as a variation of the Max Flow problem (Martel, 1982). Note that rule (vi) of Def. 2.7 is a non-issue for unicore systems. However, the hyperperiod length, and therefore the number of jobs in the hyperperiod, can be an exponential function of the number of tasks in the system; if all task periods are relatively prime, then the hyperperiod will be the product of all individual task periods. Fortunately, harmonic task systems are common in safety-critical real-time systems where CE scheduling may be called for. In this case, the largest period in the system is the hyperperiod.

In an idealized model where job executions can be preempted at any point without adding to the execution time—frame boundaries are effectively preemptions—frames can be arbitrarily small. In a more practical model, larger frame sizes that minimize the number of preemptions may be preferable. Determining acceptable frame sizes with all factors taken into account is beyond the scope of our present work.

On multicore systems, the basic idea of CE scheduling is the same: create a repeatable assignment of jobs to frames so that all jobs can complete by their deadlines. Rule (vi) becomes a concern in this case, and determining CE schedulability on a multicore system is NP hard (Burns et al., 2015).

Example 2.1. Figure 2.1 shows a possible CE schedule on two cores for a task system consisting of tasks $\tau_1 = (2,3)$, $\tau_2 = (5,6)$, and $\tau_3 = (3,6)$ with a frame size of two. Jobs $\tau_{2,1}$ and $\tau_{3,1}$ are both split across multiple frames; the final frame includes portions of jobs $\tau_{2,1}$ and $\tau_{3,1}$ on a single core. This task system can also be scheduled with a frame size of one or three. ◊

While scheduling SRT systems with this method is possible, it is not typically done. We do not give separate correctness conditions for SRT systems using this algorithm.

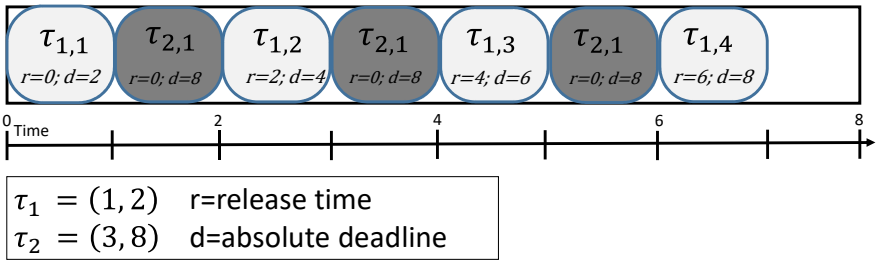

Figure 2.2: Task system scheduled with EDF.

Despite the frequent overheads caused by frame boundaries and the potentially exponential complexity, CE scheduling is often the algorithm of choice for safety-critical systems due to the high level of control it offers. With CE scheduling, it can be known in advance which jobs will execute at the same time, knowledge that can be useful in avoiding cache eviction and other means by which jobs on different cores can influence one another's execution times. Recent work on CE scheduling for multicore platforms has focused on mixed-criticalty scheduling (Burns et al., 2015; Burns and Baruah, 2017) and more efficient schedule computation (Deutschbein et al., 2019).

2.3.3 Earliest Deadline First Scheduling

EDF scheduling, as the name implies, gives highest priority to the job with the earliest absolute deadline. We first consider EDF scheduling on unicore systems. Although our work considers multicore systems, unicore EDF provides a theoretical building block used in multicore schedulers.

On a unicore system, fully preemptive EDF is optimal for both HRT and SRT systems .

Theorem 2.1. *(Liu and Layland, 1973) A task system can be correctly scheduled using EDF on a single core if and only if*

$$U \leq 1$$

holds. This result holds for both sporadic and periodic systems.

Example 2.2. Let τ be a periodic system of two tasks, $\tau_1 = (1,2)$ and $\tau_2 = (3,8)$. We have

$$U = \frac{1}{2} + \frac{3}{8} = 0.875.$$

EDF schedules τ without deadline misses; see Figure 2.2. Multiple jobs of τ_1 preempt $\tau_{2,1}$. $\Diamond$

Non-preemptive EDF. In practice, not all tasks are fully preemptable. For some task systems, the associated cache affinity loss may be unacceptable. System calls may be non-preemptable, and non-preemptivity is one means of enforcing mutually exclusive critical sections; one example is the kernelized monitors proposed by Mok (Mok, 1983). In some cases, which we consider in Chapter 5, tasks using SMT should be non-preemptable. For these reasons, it is important to consider the schedulability of systems that include non-preemptive elements. For SRT systems, the condition of Thm. 2.1 remains the same, and Non-Preempteive EDF (NP-EDF) remains optimal, even when some tasks include segments that are not preemptable (Devi and Anderson, 2005). However, a different condition is needed for HRT systems that include non-preemptive segments.

A uniprocessor EDF schedulability test that accounts for non-preemptable sections within otherwise preemptable tasks is given by Liu (Liu, 2000).

Definition 2.8. Let τ_i's *blocking term* b_i be the maximum total time for which a job of task τ_i may be prevented from executing by lower-priority jobs. ◄

More specifically, if b'_i gives the length of the maximum non-preemptable segment of any task τ_j such that $T_j > T_i$, then

$$b_i = b'_i - \varepsilon$$

holds. The worst-case scenario for τ_i occurs when τ_j begins its non-preemptable segment immediately before τ_i releases a job. If no preemptions at all are allowed, then this model can be used by defining the entirety of each task as a non-preemptable segment.

Theorem 2.2. *(Liu, 2000) (Chapter 6) Scheduling τ via EDF on a uniprocessor will result in all deadlines being met if*

$$\sum_{k=1}^{n} u_k + \frac{b_i}{T_i} \leq 1 \tag{2.6}$$

holds for all $\tau_i \in \tau$.

Example 2.3. Scheduling a periodic task system consisting of $\tau_1 = (1,2)$ and $\tau_2 = (3,8)$—the same system as in Example 2.2—using NP-EDF will cause job $\tau_{1,2}$ to miss its deadline. The resulting schedule is depicted in Figure 2.3. While this deadline miss means the schedule is incorrect for HRT, the schedule is still correct for an SRT system; no job will miss its deadline by more than a single time unit. ◊

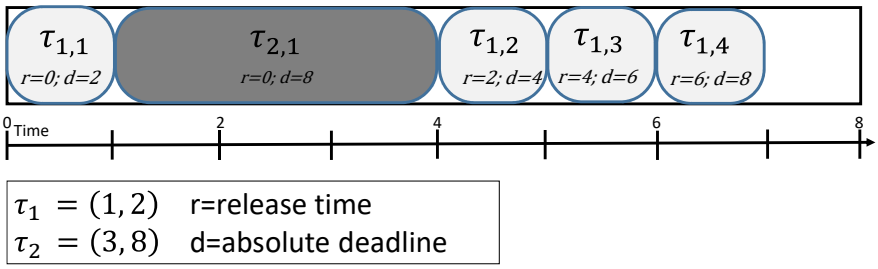

Figure 2.3: Task system scheduled with NP-EDF.

2.3.3.1 Partitioned EDF

Perhaps the most intuitive means to apply EDF to multicore systems is partitioned EDF, or P-EDF. In this algorithm, tasks are first assigned to individual cores and then scheduled using unicore EDF scheduling so long as all tasks assigned to each core have a total utilization no greater than one. Partitioning is generally similar to bin-packing and can be done using bin-packing-inspired algorithms. While this approach seems simple, it is at least NP-hard in the strong sense, and in some cases much harder (Ekberg and Baruah, 2021).

P-EDF is nearly as predictable as CE scheduling; the difference is that P-EDF does not allow control over what tasks are executing at the same time on different cores. Like CE scheduling, P-EDF can eliminate cache affinity loss due to job migration between cores.

Unlike CE scheduling, P-EDF does not require building an entirely new schedule for every change in the target system. Additionally, P-EDF is easier to use with sporadic systems than CE scheduling.

The disadvantage for P-EDF is that the number of cores required to schedule a system may be significantly more than the total utilization. If a system consists of n tasks each with $u_i > 0.5$, n cores are required to correctly schedule it. This result holds for both SRT and HRT systems.

Example 2.4. Consider scheduling the system from Example 2.1—$\tau_1 = (2,3), \tau_2 = (5,6), \tau_3 = (3,6)$—using G-EDF. The system has a total utilization of

$$\frac{2}{3} + \frac{5}{6} + \frac{3}{6} = 2.$$

As shown in Example 2.1 and Figure 2.1, it can be correctly scheduled on two cores. However, no two tasks can be scheduled entirely on the same core, as any two tasks have a combined utilization greater than one. Consequently, three cores, one per task, are required. This scenario is depicted in Figure 2.4. ◊

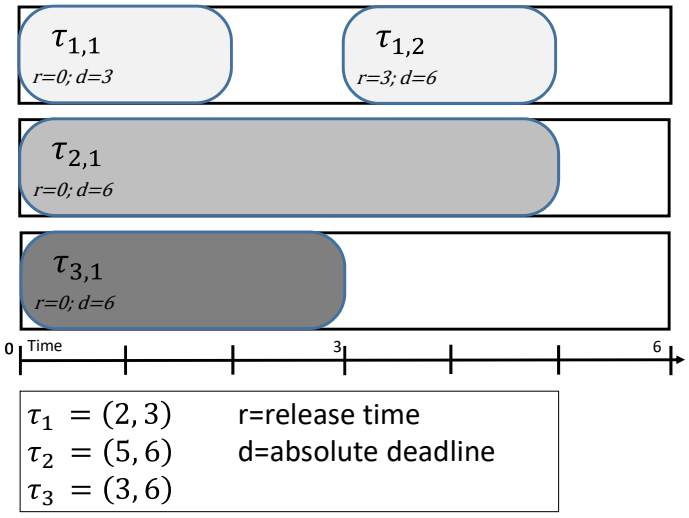

Figure 2.4: Example of P-EDF scheduling.

The worst case for P-EDF occurs when all tasks have a utilization of slightly more than 0.5. In this scenario, only one task can be assigned to each core; compared to an optimal scheduler, nearly half the available hardware goes unused.

2.3.3.2 Global EDF

Global EDF (G-EDF) again gives highest priority to jobs with earlier deadlines, but allows any job to be scheduled on any core.

Example 2.5. Figure 2.5 shows the scheduling of a three-task system—$\tau_1 = (2,4), \tau_2 = (5,8), \tau_3 = (4,10)$—on two cores with G-EDF. Deadline ties are broken in favor of the task with the lower index. Note that after $\tau_{3,1}$ is preempted, it resumes execution on a different core. ◊

G-EDF is optimal for SRT systems, i.e. any system for which $U \leq m$ holds and u_i is at most one for all tasks will be correctly scheduled (Devi and Anderson, 2005). For HRT systems, G-EDF is not optimal. In fact, for any hardware platform with at least two cores, there exist task systems with utilization arbitrarily close to one that cannot be scheduled using G-EDF (Dhall and Liu, 1978); this is known as the *Dhall Effect*.

Example 2.6. Consider a task system with m tasks for which $\tau_i = (2\varepsilon, 1)$ and one task for which $\tau_i = (1, 1+\varepsilon.)$ This system has a total utilization of

$$m \cdot 2\varepsilon + \frac{1}{1+\varepsilon} \approx 1.$$

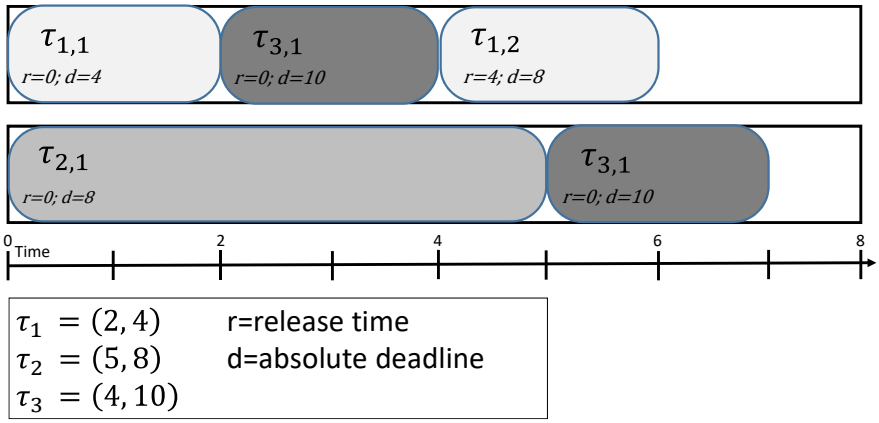

Figure 2.5: Example of G-EDF scheduling

However, correctly scheduling this system on m cores with G-EDF is impossible for HRT. Figure 2.6 depicts such a system with $m = 2$ and three tasks; the first two jobs of τ_3 both miss their deadlines. However, the system as scheduled does satisfy the bounded tardiness condition for SRT. The same system could be scheduled correctly for HRT with P-EDF on two cores by assigning the cost one task to one core all other tasks to the second core. ◊

For HRT systems, G-EDF scheduling is generally less effective than either CE or P-EDF scheduling due to both the Dhall effect and the increased difficulty of determining exact execution times when tasks can potentially be scheduled on any core.

2.3.3.3 Clustered EDF

Clustered EDF (C-EDF) combines aspects of G-EDF and P-EDF. In clustered scheduling, tasks are first assigned to a cluster of cores and then scheduled "globally" within each cluster among cores in that cluster. The problem of assigning cores to clusters is similar to the partitioned scheduling problem, but with much less capacity loss for SRT systems.

Recall that in P-EDF, every core can be assigned tasks with total maximum utilization of one. When using C-EDF for SRT, a cluster can be assigned tasks with total utilization equal to the cluster size.

Example 2.7. Consider scheduling a SRT system of twelve tasks, each with $u_i = \frac{2}{3}$. Under P-EDF, twelve cores would be required. However, with four-core clusters, each cluster could support total utilization of four, i.e. six tasks. Therefore, only eight total cores would be required. ◊

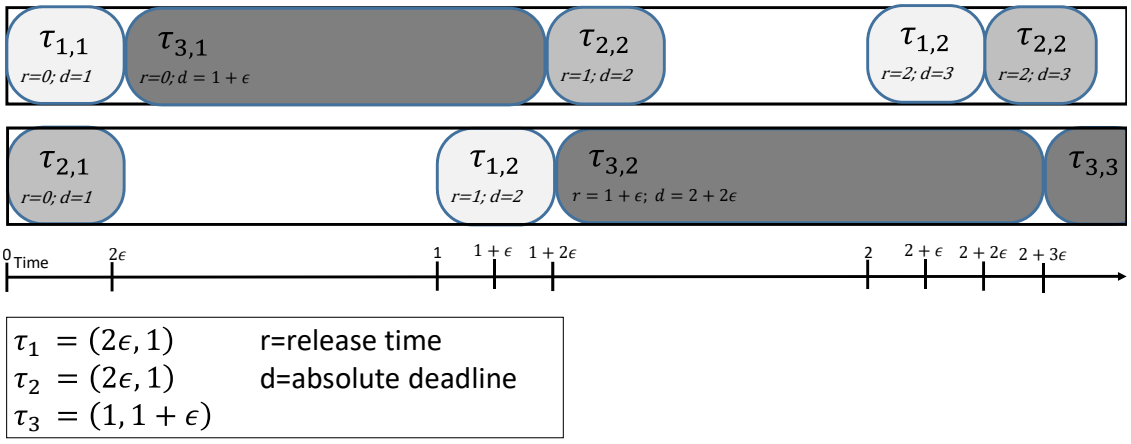

Figure 2.6: The Dhall Effect with G-EDF Scheduling.

Example 2.7 demonstrates one reason to use C-EDF: a given system may require fewer cores with C-EDF than with P-EDF. The second reason is to reduce the cost of cache affinity loss compared to G-EDF. When a job is preempted in unicore EDF, it is guaranteed to resume its execution on the same processor; this happens to $\tau_{2,1}$ in Figure 2.2. The same is true within P-EDF. In both cases, it is possible that the preempted job's data will remain in cache.

With G-EDF, however, a preempted job can resume on any core. While Figure 2.5 depicts only two cores, the preempted job $\tau_{3,1}$ could resume on any core if the three tasks in Figure 2.5 were only part of a larger system scheduled globally across many cores.

Consequently, task costs must account for the possibility of all data having to be re-acquired from main memory. Recall from Section 2.1 that while an L1 or L2 cache reference will typically take less than 10 ns, a single main memory reference can take 50-100 ns. C-EDF mitigates this problem by restricting each task to a subset of cores. If all cores in each cluster share a common *lowest-level cache* (LLC), then it is possible to provide assurances that a preempted job will at least partially retain its LLC affinity. The actual time lost to cache affinity problems will vary with the task's memory usage; tasks with higher memory use will suffer more from a loss of cache affinity.

Brandenburg (Brandenburg, 2011) has extensively documented the costs of context switching, cache affinity loss, and the time required for the scheduling algorithm itself to run in many different schedulers, including all three of P-EDF, G-EDF, and C-EDF. He found when these costs are accounted for, P-EDF is the best choice *provided* that the task set can be efficiently partitioned, i.e., tasks with $u_i > 0.5$ are rare. When P-EDF does not work well, Brandenburg found that G-EDF was preferred for tasks with low memory requirements and C-EDF for tasks with larger memory requirements.

2.3.3.4 Partially Available Cores

So far, we have been assuming that every core supporting a task system executes work belonging only to that task system. However, in Chapter 4 we will allow one core per platform to alternate between scheduling tasks that do and do not use SMT. Our analysis incorporates existing work on partially available cores.

Devi and Anderson's algorithm *EDF-high-low* (EDF-HL) shows how to schedule an SRT system of "low" priority tasks where each core may also be required to schedule at most one "high" priority task (Devi and Anderson, 2006). High tasks are always prioritized over low tasks; from the perspective of the low tasks, a core is available only when no high task is executing. Low tasks are scheduled globally across all cores, subject to any lack of availability caused by the high tasks.

Example 2.8. Figure 2.7 depicts two cores from the perspective of a system of low tasks. The first core has limited availability due to the higher priority task $\tau_1^H = (1,4)$. Black boxes indicate the times when the core is unavailable. The second core supports no high tasks and is therefore fully available for low work. $\Diamond$

Example 2.9. Figure 2.8 shows the EDF-HL schedule for a system of three low tasks, $\tau_1 = (2,4), \tau_2 = (5,8)$, and $\tau_3 = (4,10)$ subject to interruption by a single high task $\tau_1^H = (1,4)$. Note that the low portion of this task system is the same as in Example 2.5 and Figure 2.5. $\Diamond$

In our work, a core that can support tasks using or not using SMT will be unavailable to whatever group of tasks it is not currently supporting. Analytically, we can view the time this core spends not supporting one group of tasks as an interruption from a high task. In Figure 2.8, τ_1^h is equivalent to time the core spends executing tasks that use SMT and being unavailable to tasks that do not use SMT. We discuss this idea in more detail in Chapter 4.

The following definitions are used in connection with EDF-HL.

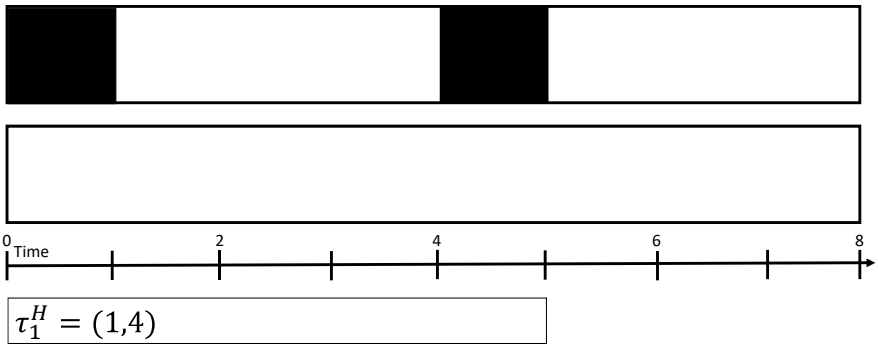

Figure 2.7: Cores with limited and full availability.

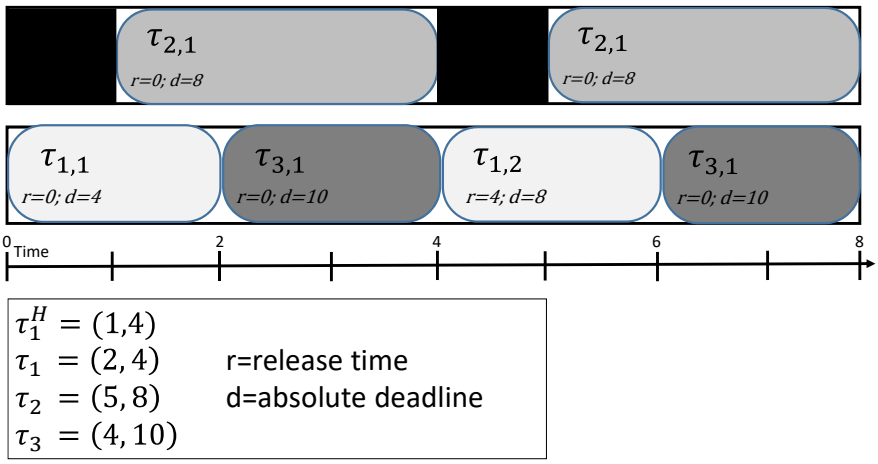

Figure 2.8: Scheduling with EDF-HL.

Definition 2.9. Let τ_H be the set of all high tasks, τ_L be the set of all low tasks, $u_{max}(\tau_L)$ the highest-utilization task within τ_L, U_{sum} be the total utilization of both τ_H and τ_L, U_H be the sum of all the utilizations of all tasks in τ_H, n_L be the number of tasks in τ_L, and U_L be the sum of the $min(\lceil U_{sum} \rceil - 2, n_L)$ largest utilization of tasks in τ_L.

Devi and Anderson have shown the following:

Theorem 2.3. *(Devi and Anderson, 2006) EDF-HL guarantees bounded tardiness to every task τ_i of τ_L if $|\tau_H| \leq m$, $U_{sum} \leq m$, and at least one of (4.3) or (4.4) holds.*

$$m - |\tau_H| - U_L > 0 \tag{2.7}$$

$$m - \max(|\tau_H| - 1, 0) \cdot u_{max}(\tau_L) - U_L - U_H > 0 \tag{2.8}$$

In Example 2.9, $|\tau_h| = 1$ and $U_{sum} = 1.775$. $\tau_H \leq m$ and $U_{sum} \leq m$ both hold. Furthermore,

$$U_L = \min(\lceil 1.775 \rceil - 2, 3) = 0.$$

Expression 4.3 holds:

$$m - |\tau_H| - U_L > 0$$
$$2 - 1 - 0 > 0.$$

It follows that the system of Example 2.9 is schedulable with bounded tardiness.

2.3.3.5 Probabilistic Correctness and Average Costs

The algorithms above use the assumption that for all tasks, C_i upper-bounds the WCET. In that case, a job will always compete once it has executed for C_i time units.

Rather than assuming C_i upper-bounds WCETs, C_i can be used as a per-job execution budget. If a job does not complete after C_i units of execution time, the task is prohibited from receiving additional processor time until it receives additional budget. C_i units of budget are allocated when (1) T_i time units have passed since the last job release and (2) there is a released but incomplete job of τ_i. Essentially, a job of τ_i that does not complete within its own budget will borrow time from the next job of τ_i.

Example 2.10. Let τ consist of two tasks, $\tau_1 = (1,2)$ and $\tau_2 = (3,8)$. These parameters are the same as the task system of Example 2.2 and Figure 2.2. The difference is that here, C_i is a budget rather than a WCET. In this example, the first four jobs of τ_1 have the actual costs of $C_{1,1} = 1.5$, $C_{1,2} = 1.0$, $C_{1,3} = 0.5$, and $C_{1,4} = 1.0$ while job $\tau_{2,1}$ has cost $C_{2,1} = 3$. The resulting schedule is shown in Figure 2.9 While $\tau_{1,1}$ overruns its execution budget, it still executes only in time that was allocated to τ_1; τ_2 is not affected. The long runtime for $\tau_{1,1}$ pushes back the execution of $\tau_{1,2}$, causing it to be late as well, despite requiring only its budgeted execution time of 1.0. However, $\tau_{1,3}$ requires less time than budgeted—-0.5—which allows both it and $\tau_{1,4}$ to complete on time. ◊

In statistics, a *random variable* takes on a value dependent on random events. For example, the random variable D might be used to represent the possible outcomes of rolling a six-sided die; potential values of D would be the integers one through six. A random variable's *expected value* is the average of all possible

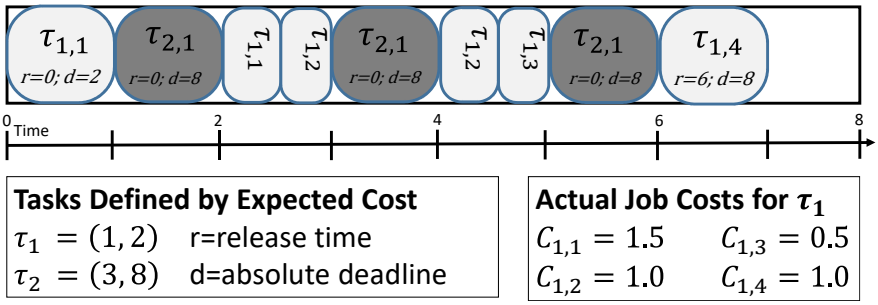

Figure 2.9: C_i used as a budget within an EDF schedule.

outcomes weighted by the probability of each value occurring. If D represents outcomes on a fair die, its expected value would be 3.5. If the die were weighted so that half of all rolls would give a six and the probability of rolling each number from one through five were $\frac{1}{10}$, the the expected value of D would be 4.5.

If we view a task's execution times as a random variable, then the expected value of a task's execution time is equivalent to its ACET, assuming that "average" refers to the average of *all* execution times and not merely the observed execution times. Since an equivalent assumption is in place for WCETs—a task's WCET is its worst possible execution time, not merely its worst observed execution time—we are justified in this assumption.

Mills and Anderson (Mills and Anderson, 2011) have shown that if (1) C_i is used as budget as in Example 2.10; (2) for all tasks, C_i is greater than the expected execution time; and (3) τ as an SRT system would be strictly correct if C_i upper-bounded WCETs, then τ is probabilistically correct per Def. 2.5.

Defining C_i in this way may be useful when WCETs are unknown—realistically, this is always the case in multicore systems with unpredictable cross-core task interactions (Wilhelm, 2020)—or orders of magnitude larger than ACETs.

2.4 Directed Acyclic Graphs

The scheduling discussions above apply to tasks where each job must be executed sequentially. However, as already discussed in Section 1.5.3.1, parallelism is an essential part of many modern applications and is widely modeled using DAGs.

The basic idea of the DAG task model was given in Section 1.5.3.1. Here we introduce some additional terminology for the DAG model and a brief survey of existing scheduling approaches. We duplicate Figure 1.2 from Section 1.5.3.1 for easy reference.

2.4.1 Model Overview

A DAG task is defined as $\tau_i = (G_i, T_i)$, where $G_i = (V_i, E_i)$ is a DAG. T_i, as in the standard sporadic task model, gives the period of τ_i, which releases a *DAG job* at most once every T_i time units, beginning at time 0. As before, we assume *implicit deadlines*: every DAG job must complete within T_i time units of its release.

DAG-specific terminology. Each DAG task τ_i consists of $|V_i|$ subtasks, with each subtask corresponding to a vertex in G_i. The *cost* of subtask v_k is given by c_k, assumed to be its WCET. The sum of all subtask costs gives the task's *total cost* C_i. We use capital letters to refer to characteristics of tasks and lower-case letters for characteristics of subtasks. When unambiguous, we omit subscripts from the capital letters. Each DAG job consists of one subjob for each subtask in the DAG. The DAG job is complete once all component subjobs have completed.

If within G there exists a path from vertex v_1 to v_2, then v_1 is a *predecessor* of v_2 and v_2 is a *successor* of v_1. Otherwise, they are *unconnected* and the corresponding subjobs within a single job can execute in any order, including simultaneously. If the vertices connect via a one-edge path, then the two are *immediate* predecessors and successors. We require that subtask indices follow a topological order, i.e., if $i < j$ holds, then v_j is not a predecessor of v_i.

Definition 2.10. A *chain* is a sequence of vertices in which all but the last are followed by one immediate successor. ◄

Definition 2.11. A DAG's total *length* L is equal to the maximum length of any chain in G. This value gives the minimum amount of time required to fully execute all jobs given an unlimited number of processors. ◄

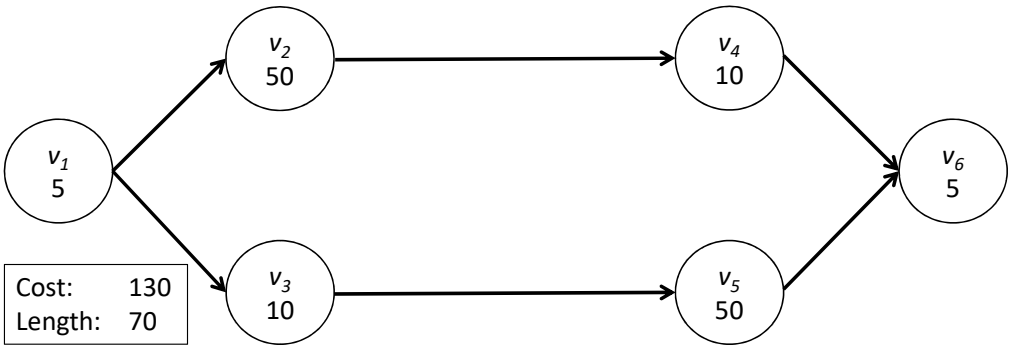

Figure 2.10: A DAG task consisting of six subtasks. Duplicate of Fig. 1.2

.

Example 2.11. In Figure 2.10, the sum of all subtask costs gives the DAG's cost of 130. The two longest chains in the DAG are v_1, v_2, v_4, v_6 and v_1, v_3, v_5, v_6. Both chains have a length of 70, giving the DAG $L = 70$. ◊

Length is the formal term for the minimum task completion time of 70 without SMT in Examples 1.2 and 2.11. We require that $L \leq T$ holds for all tasks; otherwise, the task cannot be scheduled. While the term length is common in the DAG-scheduling literature, some sources use the term *critical path.*

A task's total *utilization* is defined as $U = \frac{C}{T}$. While purely sequential tasks must have $U \leq 1$ to be schedulable, DAG tasks with $U > 1$ are schedulable, given a sufficient number of cores, if $L \leq T$ holds (Baruah et al., 2012). We differentiate between *heavy* and *light* tasks.

Definition 2.12. τ_i is *heavy* if $U > 1$ holds and *light* otherwise. ◄

While light tasks can be scheduled sequentially on a single core without parallelism, heavy tasks can be scheduled only by executing subtasks in parallel across multiple cores.

2.4.2 Scheduling Single Heavy DAGs

Even without SMT, determining the minimum number of cores needed to schedule a heavy DAG is NP-hard in the strong sense (Baruah, 2015) (light tasks can be scheduled sequentially on one core). To determine how many cores are needed by a heavy task, we follow Baruah (Baruah, 2015) by using Algorithm 1. In this algorithm, a job begins execution as soon as all of its predecessors have been allocated their WCETs and at least one core is free.

The maximum number of cores that a greedy algorithm such as Algorithm 1 will assign to a given DAG is bounded.

Theorem 2.4. *(Li et al., 2014) If an implicit-deadline deterministic parallel task is assigned*

$$m = \left\lceil \frac{C-L}{T-L} \right\rceil \tag{2.9}$$

dedicated cores by a greedy algorithm, then all jobs will meet the relative deadline of T.

Expression 2.9 is undefined when $T = L$. However,

$$m = \frac{C-L}{\varepsilon} + 1 \tag{2.10}$$

Algorithm 1 Assign Cores (Baruah, 2015); incorporates elements from (Graham, 1969)

```
Maintain ready as the set of all subjobs without incomplete predecessors.
Maintain m_r as the number of cores not currently executing a subjob.
m = ⌈U⌉
Let t be the time and D the DAG deadline.
while True do
    t = 0
    m_r ← m
    ready ← subjobs without predecessors
    while t ≤ D do
        while ready ≠ ∅ ∧ m_r > 0 do
            Assign ready subjobs to cores.
        end while
        t ← next subjob completion time
        if all subjobs are complete and t ≤ D then
            {m-core schedule found.}
            return m
        end if
    end while
    {No m-core schedule found; increment m.}
    m ← m + 1
end while
```

is a necessary and sufficient condition to schedule a DAG given C, L, and T parameters but arbitrary internal structure (Li et al., 2014).

Example 2.12. Figure 2.11 shows a DAG-task with $C = 10$, $L = 6$, and $T = 7$. As predicted by Theorem 2.4, it can be scheduled to meet its deadline on four cores. In this case, four cores is optimal. ♢

It follows from Theorem 2.4 that if $T > L$ holds, then the outermost loop of Algorithm 1 will have at most

$$\left\lceil \frac{C-L}{T-L} \right\rceil - \lceil U \rceil + 1.$$

iterations. If $T = L$, then the outermost loop will have at most

$$\left\lceil \frac{C-L}{\varepsilon} \right\rceil - \lceil U \rceil + 2$$

iterations. In both cases, Algorithm 1 first attempts to find a schedule on $\lceil U \rceil$ cores (the minimum possible number) and increments m until a correct schedule is found; the maximum number of iterations needed is one more than the difference between the maximum number of cores needed and $\lceil U \rceil$.

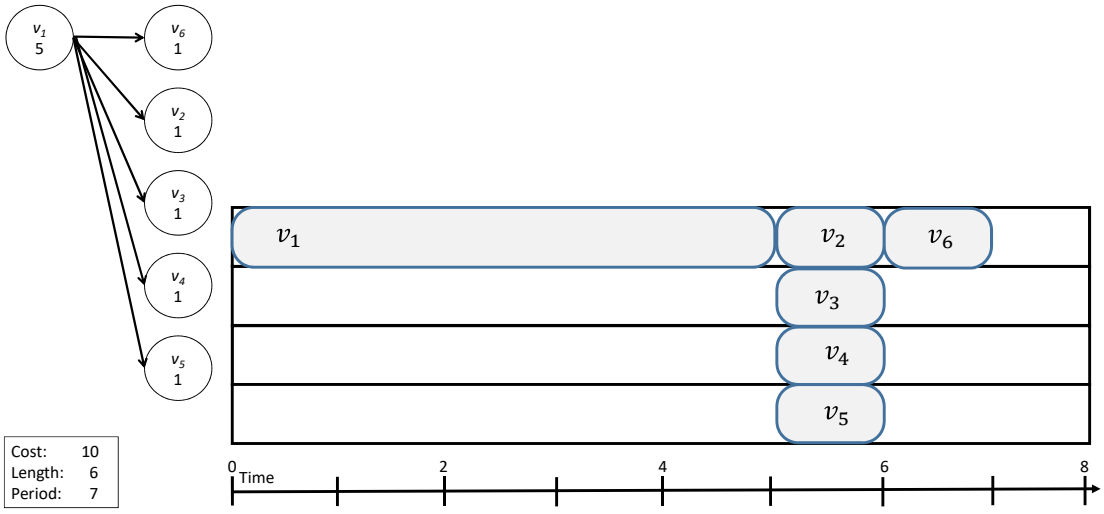

Figure 2.11: A DAG task scheduled on four cores.

Theorem (2.4) is pessimistic; it considers only the task parameters C, L, and T. Depending on the exact structure of the DAG, fewer cores may be possible.

Example 2.13. Figure 2.12 shows a second DAG-task with $C = 10$, $L = 6$, and $T = 7$. Although these parameters are the same is those of Ex. 2.12, this task can be scheduled via Algorithm 1 on only two cores. While this result is optimal, the algorithm is not: if v_2, v_3, v_4, and v_5 are all scheduled before v_1, then Algorithm 1 would assign fail to find a two-core schedule and would instead assign three cores. ◊

Algorithm 1 has a speedup bound of

$$b = 2 - \frac{1}{m} \tag{2.11}$$

i.e. if an optimal algorithm can schedule the DAG on m unit-speed cores, then Algorithm 1 can schedule the same DAG on m speed b cores (Baruah, 2015). An optimal algorithm would, by definition, have a speedup bound of 1.0. Informally, this result shows that using Algorithm 1 instead of an optimal algorithm has an effect no worse than using the optimal algorithm on half-speed cores.

The bound of Theorem 2.4 was recently improved by He et al. (He et al., 2022) through means of considering many chains within the DAG rather than only the DAG's length, total cost, and period; the authors found experimentally that their bound could call for as few as 20% of the cores called for by Theorem 2.4.

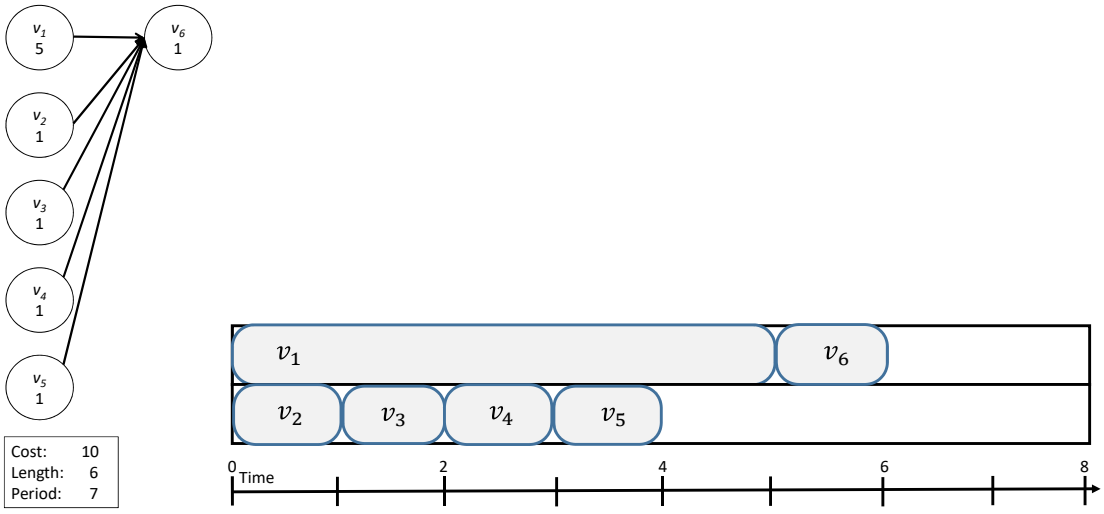

Figure 2.12: A DAG task scheduled on two cores.

2.4.3 Federated Scheduling and Related Methods

In research on scheduling systems of DAGs, federated scheduling on shared hardware is a prominent and well-studied approach. In this method, a system of DAG tasks is scheduled by executing each heavy task on a set of dedicated cores and partitioning light tasks among the remaining cores (Li et al., 2014). Heavy tasks may be over-assigned capacity; for example, the task of Ex. 2.12 and Figure 2.11 requires four cores despite having a utilization of only $\frac{10}{7} \approx 1.43$.

Despite the potential capacity loss of federated scheduling, it has generally good theoretical performance, low scheduling overheads, and avoids cache affinity loss (Li et al., 2022). For these reasons, we use federated scheduling as the basis for our work in Chapter 5 on SMT and DAGs.

For soft real-time tasks, where some deadline misses are allowable, cores can be reclaimed by work stealing, where idle cores "steal" work from busy ones (Li et al., 2016). Other methods to reduce the capacity loss of federated scheduling include semi-federated scheduling (Jiang et al., 2017) and reservation-based federated scheduling (Ueter et al., 2018). In semi-federated scheduling, portions of heavy tasks are scheduled as if they were light tasks; the task of Ex. 2.12 and Figure 2.11 that would be assigned three cores under federated scheduling would be assigned one dedicated cores under semi-federated scheduling, with the remaining portion of the task scheduled independently. In reservation-based federated scheduling, DAG tasks are assigned reservation servers that can be scheduled sequentially.

2.5 Timing Analysis

So far, we have considered *scheduling* analysis but not *timing* analysis; all of the schedulability results in Sec. 2.2 require that for all tasks τ_i, C_i upper-bounds either its WCET or ACET. Here, we show why that assumption is non-trivial and discuss existing means to estimate execution costs.

Historically, there has been a separation of concerns in the real-time community between timing analysis (determining a task's execution time) and schedulability analysis (determining whether a set of tasks with known execution times can be correctly scheduled). There are good reasons for this separation: both topics are complex on their own, making it difficult to address both within a single work.

2.5.1 Difficulties of Determining WCETs

There are two problems with modeling a task based on its WCET. First, determining the true WCET may be prohibitively difficult, especially on a multicore machine. The classical approach to determining WCETs is *static timing analysis*, which relies on detailed knowledge of the task being executed and the platform executing it. When multiple tasks can interfere with one another, knowing the exact state of the platform becomes essentially impossible, preventing static analysis (Wilhelm, 2020). Possible ways a job may change the state of the platform, and hence affect the execution of other jobs, include causing cache conflicts (Altmeyer et al., 2014; Bui et al., 2008; Chisholm et al., 2015; Kirk, 1989; Mancuso et al., 2013; Ward et al., 2013; Xu et al., 2016), DRAM conflicts (Guo and Pellizzoni, 2017; Hassan et al., 2015; Reder and Becker, 2020; Valsan et al., 2016; Yun et al., 2014, 2013), memory bus conflicts (Muench et al., 2014; Seetanadi et al., 2017; Xu et al., 2019), and I/O conflicts (Kim et al., 2014; Pellizzoni et al., 2008). A recent survey on the difficulties associated with WCET verification has been presented by Maiza et al. (Maiza et al., 2019). Industrial-quality WCET analysis requires both sophisticated tools and expert analysis; the process cannot be entirely automated (Rapita Systems, 2021).

Second, if the WCET can be determined, it may be a very rare event that a job actually requires that many units of execution time, leading to very pessimistic scheduling if it is assumed that every job of τ_i will require its WCET; depending on the application, this pessimism may be unnecessary.

One possible alternative to finding WCETs is *probabilistic worst-case exeuction times,* or pWCETs.

Definition 2.13. E_i is a random variable equal to the execution time of a randomly selected future job of τ_i.

◄

Definition 2.14. τ_i has the *pWCET* C_i^p if and only if execution times for τ_i follow a probability distribution[2] such that

$$\Pr(E_i \leq C_i^p) = p \tag{2.12}$$

holds. We refer to p as the *provisioning level* of τ_i. ◀

However, finding a task's pWCET has its own problems. Validating that C_i^p is a pWCET for a task τ_i requires a detailed understanding of the task's behavior in many possible execution states and of the likelihood of those states occurring. It can be argued that determining a pWCET is no more practical than determining a WCET. Furthermore, it is potentially dangerous to determine a pWCET based only on observed run-times. To understand why, consider a simpler problem: rolling a die. Finding exact probabilities of a given die roll result, which is analogous to a pWCET, requires detailed knowledge of the die's physical properties, which is analogous to perfect knowledge of a task's behavior. Lacking detailed knowledge of a die or task, we can make predictions based on observed past rolls or execution times. This method is practical, but does not produce a pWCET for the task.

Example 2.14. We wish to know $\Pr(\text{an arbitrary future roll} \leq 5)$ for a given die, but we do not know any of the die's properties, such as how many sides it has or whether it is balanced. We cannot examine the die directly; we can only tabulate rolls. Suppose that out of ten roles, the maximum was five. Any of the following are possible: the die has no faces greater than five, the die has faces greater than five but these are unlikely to be rolled, or the die has faces greater than five, which are reasonably likely to be rolled, but our set of ten rolls was not representative of the die's true long term behavior. To illustrate the last case, rolling a fair six-sided die ten times gives

$$\Pr(\text{ten rolls} \leq 5 \mid \text{a fair six-sided die}) = \left(\frac{5}{6}\right)^{10} \approx 0.162.$$

Consequently, a naïve interpretation of observed results, such as the rule that getting zero sixes out of ten rolls means a six is less likely than other roles, will frequently be wrong. ◊

Inferring probabilities from measured random data forms the basis of a family of methods known as Measurement-Based Probabilistic Timing Analysis (MBPTA). These methods attempt to probabilistically bound task execution times by examining a small sample of jobs (Cazorla et al., 2019; Davis and Cucu-

[2]In some sources, the distribution itself is referred to as the pWCET

Grosjean, 2019). MBPTA was first proposed by Burns and Edgar in 2000 (Burns and Edgar, 2000), in part as a means to address the difficulty of timing analysis on complex processors; concerns over the practicality of static timing analysis predate multicore computing. With MBPTA, each trial is used to gather data, similarly to each roll of a die. A set of execution times is called a *trace*.

Definition 2.15. A trace is an ordered set of consecutive execution times for a task and a finite-sized **s**ample from the population of all possible execution times for that task. For trace A with $|A|$ elements, let $\{A_1, A_2, ...A_{|A|}\}$ be the ordered sample times within the trace. Let the maximum of the trace be denoted by A_{max}. Note that the A_k values refer to times within a finite-sized sample, while E_i is from the entire population of execution times. ◄

The trace is analyzed to produce an estimate of future behavior. The difficulty is that measured values are themselves random variables. Statements of the form

$$\Pr(\text{arbitrary future roll or execution time} \leq Y) \geq p$$

cannot be correctly evaluated without the understanding that Y, when based on measurement data, is itself a random variable. An additional challenge is that unlike dice rolls, execution times cannot be assumed to be independent.

In statistics, results of a repeated random trial are *independent* if and only if the results of one trial reveal nothing about the results of any other trial. Dice rolls are independent: when rolling a die, each roll reveals nothing about the next roll. Other examples of independent events include repeated coin flips and roulette wheel spins. Independence is not the same as having equal-probability outcomes; even if a coin is weighted to land on heads with probability 0.75 the probability of the next flip being heads is the same regardless of previous flips.

Unfortunately, execution times are not independent; if the execution time of one job is exceptionally long, the probability that the next job's execution time will also be long often increases. Safely addressing these factors—measurements are themselves potentially random, but not necessarily independent—lies at the heart of MBPTA analysis.

A concept related to independence is *identical distribution*. Random events are identically distributed if and only if the same underlying probability distribution applies to every event. For example, a series of dice rolls is not identically distributed if a different die is used for each roll.

The ideal situation for analysis is that random trials—equivalently, random variables—are *independent and identically distributed* (IID). A vast number of statistical results, including several we consider in this section and Chapter 3 hold only for IID random variables.

2.5.2 Extreme Value Theory

The majority of recent MBPTA work has focused on a family of methods known as *extreme value theory (EVT)*. Although we do not use EVT ourselves, a brief discussion provides insight into the difficulties associated with timing analysis. The use of EVT in real-time scheduling predates the ubiquity of multicore scheduling; it was first considered in the field to aid in determining execution times on unicore platforms (Burns and Edgar, 2000).

EVT is built around the Fisher-Tippett-Gnedenko Theorem (Fisher and Tippett, 1928):

Theorem 2.5. *Let X_1, X_2, ...X_n be a sequence of IID random variables with cumulative distribution function F. Suppose there exist two sequences of real numbers $a_n > 0$ and $b_n \in \mathbb{R}$ such that the following limits converge to a non-degenerate distribution function:*

$$\lim_{n\to\infty} \Pr\left(\frac{\max X_1, ..., X_n - b_n}{a_n} \leq X\right) = G(x).$$

Then the limit distribution G belongs to either the Gumbel, the Frechet, or the Weibull family.

Informally, this theorem can be summarized by saying that the extremes of independent and identically distributed random variables converge to one of three possible distributions. All three distributions are depicted in Figure 2.13.

The Gumbel distribution supports an infinite range of values, whereas the Frechet distribution has infinite support on the right only and the Weibull has infinite support on the left only. The Frechet distribution has a fatter right tail than the Gumbel distribution.

To apply EVT to execution time data, a practitioner attempts to fit one or more of the listed distributions to the extremes of a sample data set. Extremes are defined as either the set of all values greater than a specified threshold (the peak-over-threshold method, PoT) or the set of local maximums when the sample execution time data is divided into equal size blocks (the block maxima method, BM). For example, given a trace of 1000 execution times, a practioner might, in the PoT method, define the threshold as the 90th percentile and use the largest 100 values to select and parameterize an EVT distribution. Alternatively, using the BM

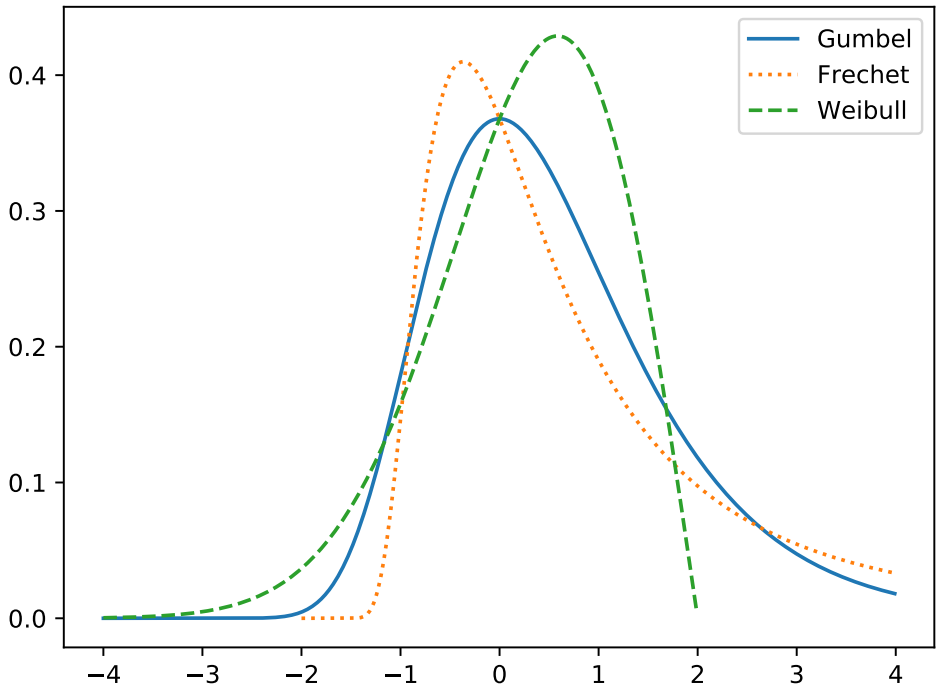

Figure 2.13: Probability density functions (pdfs) for the Gumbel, Frechet and Weibull distributions with location parameter 0 and scale parameter 1.

method, he might use the maxima of the first ten execution times, second ten, and so on. In either case, the best-fitting distribution is then used as the basis for further predictions regarding WCETs.

However, this approach contains multiple potential pitfalls. There is no universal agreement on either what block size to use or how thresholds should be set (Davis and Cucu-Grosjean, 2019). As stated in Theorem 2.5 and subsequently reaffirmed, EVT requires IID observations (Balkema and De Haan, 1974; Fisher and Tippett, 1928; Pickands, 1975), and tends to be highly sensitive to violations of this requirement. Unfortunately, real execution times frequently do include unavoidable dependencies. There is currently no agreement on how to work around this issue (Cazorla et al., 2019; Davis and Cucu-Grosjean, 2019; Jiménez Gil et al., 2017). Although research on testing the reliability of EVT is ongoing (Arcaro et al., 2020), it is in many ways still immature with regards to WCET estimation (Reghenzani et al., 2019).

2.5.3 Adding in SMT

The challenges throughout this section exist *even without SMT.* While SMT does add additional factors to be considered, we will show they are small in comparison to the many sources of timing difficulty that already exist on a multicore platform without SMT. We do not intend to solve the existing timing analysis problem, but we do show that timing analysis with SMT is comparable to timing analysis without.

2.6 Simultaneous Multithreading

In this section, we provide more details on both what SMT is and how it has been used in the past.

2.6.1 Technical Information

In many modern processors, each core uses instruction-level parallelism within jobs to execute multiple instructions per cycle. SMT builds on this behavior to allow two or more jobs to execute instructions within a single cycle. An overview of SMT execution is given in Examples 1.1 and 2.15. Further details on the fundamentals of SMT can be found in the work of Eggers et al. (Eggers et al., 1997) .

Example 2.15. Let τ_1 and τ_2 be two real-time tasks. At the top of Figure 2.14, jobs of τ_1 (darker) and τ_2 (lighter) execute sequentially without SMT on a core that can accept two instructions per cycle. When fewer than two instructions are can be executed, as in cycles three and four, cycles are wasted. τ_1 finishes at the end of six cycles and τ_2 at the end of twelve. In the second part of the figure, the same jobs employ SMT to execute in parallel, thereby reducing the number of lost cycles. τ_1 finishes after eight cycles and τ_2 after ten cycles. SMT thus delays the completion of τ_1, but speeds up the completion of τ_2 since it does not have to wait for τ_1 to complete before beginning its own execution. In the bottom portion of the figure, τ_2 begins execution first and initially has sole use of the core, with τ_1 beginning later. $\Diamond$

Figure 2.14 gives a more detailed view of SMT than Figure 1.1; whereas Figure 1.1 focuses entirely on SMT's effects on execution times, Figure 2.14 begins to show what is happening at the hardware level.

To understand SMT in more detail, some background on processor design is necessary. To that end, we list below the five steps of the classic reduced instruction set computer (RISC) pipeline (Patterson and Sequin, 1981). While many modern processors include additional steps (Fog, 2018; Hennessy and Patterson, 2019), the classic pipeline is useful to illustrate key concepts.

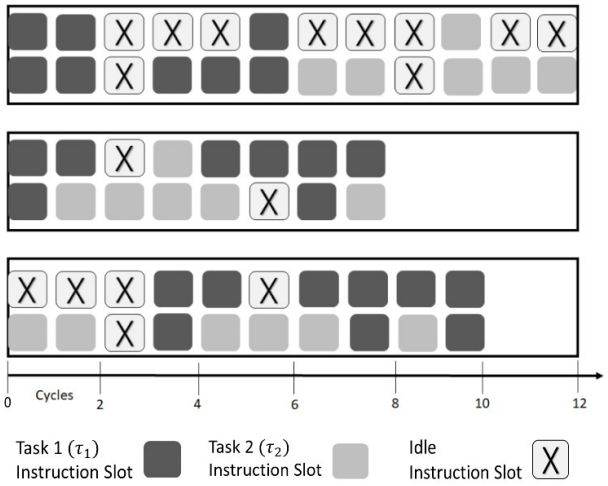

Figure 2.14:
Top: task execution without SMT.
Middle: execution with SMT.
Bottom: execution with SMT, different start times.

1. Instruction fetch. An instruction is retrieved from memory or an instruction cache.
2. Instruction decode.
3. Execute. Operations are performed using processor components such as an arithmetic logic unit (ALU) or bit shifter. If the instruction is for memory access, the effective memory address is determined.
4. Memory access. If the instruction is to load or store data, memory is read from or written to, respectively. Otherwise, this stage is skipped.
5. Writeback. Results—either from a memory load or the execution step—are written to the register file.

With pipelining in place, it is possible for one processor to complete one instruction per clock cycle even though each instruction requires multiple cycles. Ideally, every clock cycle will see one instruction at each stage of the pipeline. Modern superscalar processors enable instruction-level parallelism by including wide-issue pipelines that can process multiple instructions per pipeline phase, per clock cycle. In theory, such a processor can execute as many instructions per cycle as the pipeline is wide. The Skylake architecture, which we use in our experiments, can execute up to four instructions per clock cycle (Fog, 2018). In practice, an architecture's maximum degree of instruction-level parallelism is not always achieved. Dependencies between instructions and memory stalls can lead to a lack of executable instructions, preventing the use of available processor resources.

SMT addresses this problem by allowing each fetch to pull from multiple processes. The rate at which a task executes while using SMT is in part determined by the rate at which instructions from that task are fetched (Tullsen et al., 1996; Fog, 2018). However, early research on SMT focused on techniques designed to maximize throughput while minimizing slowdowns to any one task. For example, Tullsen et al. (Tullsen et al., 1996) proposed multiple techniques for deciding which instructions should be fetched in each cycle. These include:

- ICount: prioritize the task with the fewest instructions in the queue;
- BrCount: prioritize the task with the fewest unresolved branches.
- MissCount: prioritize the task with the fewest data cache misses.

For all three of these policies, the underlying idea is to protect tasks that make efficient use of the processor from pointless interference.

Example 2.16. Consider a processor capable of fetching a constant f instructions per cycle. Two tasks are co-scheduled on this processor. All instructions for τ_1 can progress through the pipeline at one stage per cycle once fetched. The execution speed of τ_1 is therefore limited only by the average number of instructions fetched per cycle.

τ_2's execution speed is more limited; due to a combination of branching code, data dependency, and high-latency instructions, τ_2 can writeback an average of at most w instructions per cycle no matter how quickly instructions are fetched. Allowing τ_2 to fetch more than w instructions per cycle on average will not increase the execution speed, i.e., decrease the execution cost, of τ_2 but doing so will decrease the execution speed, i.e., increase the execution cost of τ_1.

Suppose the two tasks initially have equal fetch priority. If $\frac{f}{2} < w$ holds, then enqueued instructions for τ_2 will increase over time. If the ICount policy is in use, τ_2's average instructions fetched per cycle will decrease until its average fetch is equal to w. This change benefits τ_1 without harming τ_2. $\Diamond$

ICount was found by Tulsen et al. to be the most successful in terms of maximizing throughput and has been used by subsequent works that simulate SMT-enabled processors (Eggers et al., 1997; Jain et al., 2002). ICount effectively encapsulates both BrCount and MissCount; both cache misses and unresolved instructions will lead to the presence of more unresolved instructions.

Unfortunately, chip manufactures tend to be secretive about the exact methods used to pick instructions for fetching. For this reason, we will be determining execution times with SMT using a measurement-based process. However, knowledge of potential instruction-picking criteria can help in both making rough predictions of SMT behavior and as a sanity check for observed behavior.

In general, memory-intensive programs experience minimal slowdown when using SMT, presumably because they experience high latency to begin with. In contrast, CPU-intensive programs, such as a cache-aware matrix multiplication program, tend to see their performance reduced dramatically.

2.6.2 Use Outside of Real-Time Computing

SMT became widely available in 2002 with the Pentium 4 processor (Marr et al., 2002). Numerous papers were published in the early 2000s benchmarking the average-case performance of SMT; in the majority of scenarios tested, programs utilizing SMT were found to execute at 50% to 95% of their execution speeds without SMT (Bulpin and Pratt, 2004; Bulpin, 2005; Tuck and Tullsen, 2003). Further analysis of these results seem to indicate policies that prioritize a combination of throughput and fairness. More recent work comparing execution times with and without SMT is rare; there seems to be a consensus (outside of real-time computing) that using SMT is generally the best choice and that further benchmarking is not warranted. In fact, modern machines have SMT enabled by default. SMT can be turned off by accessing advanced boot options or, in Linux, by using the command

```
sudo echo 0 > /sys/devices/system/cpu/cpuX/online
```

to turn off one processor—Linux views each hardware thread as an individual processor—per physical core with X giving appropriate processor numbers. Which processors share a physical core can be discovered by viewing system information with the command `cat/proc/cpuinfo`.

2.6.3 Past Uses in Real-Time Computing

The first attempt to utilize SMT in a real-time context was made in 2002 by Jain et al. (Jain et al., 2002), who showed that, by enabling SMT and making every thread available for real-time work, it is possible to schedule workloads with total utilizations up to 50 percent greater than what would be possible on the same platform without SMT. While Jain et al. gave ample experimental evidence that SMT can enable systems

with higher utilization to be supported, neither they nor anyone else, to our knowledge, has provided an analytic schedulability test that takes SMT into account

Since then, work on SMT in real-time systems has been sparse. One line of work has considered changing how co-scheduled tasks are prioritized against each other to ensure that a favored task has sufficient execution time; it is co-scheduled with an unfavored task that has fewer guarantees (Dorai et al., 2003; Lo et al., 2005).

Related to this work are attempts to provide a greater level of control over how different tasks using SMT are prioritized (Anantaraman et al., 2003; Cazorla et al., 2006; Gomes et al., 2016, 2015) or purpose-built processors (Zimmer et al., 2014). Unfortunately, the only hardware currently available that allows prioritization between co-scheduled threads is the IBM Power series, and even that offers less control over priority between co-scheduled tasks than would be ideal for this work.

Mische has considered using SMT to hid context-switch times by using threads to switch task state in and out in the background (Mische et al., 2010). While intriguing, this idea does not seem to have been pursued further.

2.7 Chapter Summary

In this chapter, we have set the stage for our contributions by defining the problems and tools we have to build on. We stated our assumptions regarding hardware use and gave a detailed explanation of how real-time systems can be modeled, including what it means for a system to be correct. We next gave an overview of the scheduling algorithms we will be building on: cyclic executive scheduling and earliest-deadline-first scheduling, with the latter category including unicore, global, partitioned, and clustered variants. We summarized the advantages and disadvantages of each of these approaches. We also considered the modifications to our scheduling model and additional algorithms used to scheduled tasks with internal parallelism, modeled as DAGs. We briefly touched on the difficulties of achieving a sound timing analysis in practice. Finally, we explained what SMT is and briefly described its history.

CHAPTER 3: TIMING

In this chapter, we document our experiments to observe SMT's effects on execution times within a real-time context. In addition, we show how to model these effects for the purpose of creating randomly generated synthetic task systems.

The purpose of this chapter's experiments is to gain a general understanding of SMT's effects on execution times. Strictly speaking, our conclusions only apply to one set of benchmark tasks running on a specific hardware platform using a specific operating system. If SMT is being evaluated for use in a specific application, one of the first steps in that evaluation should be running benchmark experiments with that application's tasks, hardware platform, and operating system.

In Section 3.1, we give our approach for timing and modeling the effects of SMT in the context of SRT systems. In Section 3.2, we make the case that HRT systems that use SMT are no less safe than HRT systems without SMT. In Section 3.3, we measure the effects of SMT in an HRT context and give our approach to modelling for that case. Finally, in Section 3.4 we summarize our efforts and suggest directions for future work. All experiments in this chapter used an Intel Xeon Silver 4110 2.1 GHz Skylake CPU running Linux 5.17.0. We leave the question of SMT on different architectures to future work.

3.1 Timing Analysis for SRT Systems

In this section, we present our work defining, measuring, and modeling task costs with SMT in an SRT context. We begin by briefly describing our scheduling method for SMT-aware SRT scheduling, which we will discuss in more detail in Chapter 4. We then define task costs in the context of our method and describe the experiments we conducted to understand the relationships between task costs with and without SMT. Based on our experiments, we give parameters for probability distributions that can be used to realistically model SMT's effects. This section builds on work first published in 2019 (Osborne et al., 2019). Table 3.1 summarizes the notation used in this section.

Table 3.1: Summary of notation used in Sec. 3.1

Symbol	Description	Reference	Sections Used
$C_{i(j)}$	Cost of an SRT task τ_i when co-scheduled with τ_j	Def. 3.1	3.1
τ^h	Set of all tasks with which τ_i can be co-scheduled	Def. 3.2	3.1
C_i^h	Threaded cost; $\max_{\forall\ \tau_j\ \in\ \tau^h\ :\ i \neq j}\ C_{i(j)}$	Def. 3.2	3.1
$M_{i(j)}$	SRT Multithreading Score; $\frac{C_{i(j)}-C_i}{C_i}$	Def. 3.3	3.1
V_i	Vulnerability of task τ_i	Defs. 3.4 and 3.21	3.1, 3.3
a_h	Multiplier for harmful tasks	Def. 3.5	3.1
a_s	Multiplier for standard tasks	Def. 3.5	3.1
H_j	Binary random variable indicating if τ_j is harmful	Def. 3.6	3.1
h	Probability of a task being harmful	Def. 3.6	3.1
r	$\frac{a_h}{a_s}$	Def. 3.6	3.1

3.1.1 Scheduling Method Overview

We schedule SRT tasks as if each SMT-provided thread were a separate core. We use a variation of C-EDF scheduling (see Section 2.3.3.3) in which all tasks using SMT and assigned to the same cluster can potentially be co-scheduled.

Example 3.1. Let τ consist of three tasks, $\tau_1 = (3,4)$, $\tau_2 = (5,8)$, and $\tau_3 = (4,10)$. The given costs denote each task's expected execution time when SMT is used; how to fully define costs with SMT is covered in Section 3.1.2. A possible EDF schedule is shown in Figure 3.1. While $\tau_{1,1}$ is co-scheduled only with $\tau_{2,1}$, $\tau_{2,1}$ is co-scheduled with jobs of both τ_1 and τ_3. $\tau_{3,1}$ is first co-scheduled with $\tau_{2,1}$, briefly has no co-scheduled job, and then is co-scheduled with $\tau_{1,2}$. The system is sporadic; note that $\tau_{1,2}$ does not release until time 6. Consequently, it is impossible to know in advance exactly how jobs will be co-scheduled. ◊

Assigning tasks to clusters and deciding which tasks should use SMT is covered in Chapter 4.

3.1.2 Defining an Appropriate Cost

In order to bound tardiness in a scenario similar to Example 3.1, each task's cost needs to account for all co-scheduling possibilities. Unfortunately, it is not possible to observe all possible ways in which jobs of even just two tasks may interact. To do so, we would need to test the two tasks τ_i and τ_j when starting simultaneously, when τ_i starts at any point in an already-running τ_j, and when τ_j starts at any point in an already-running τ_i. The possibilities are essentially limitless. For this reason, our experiments focus on

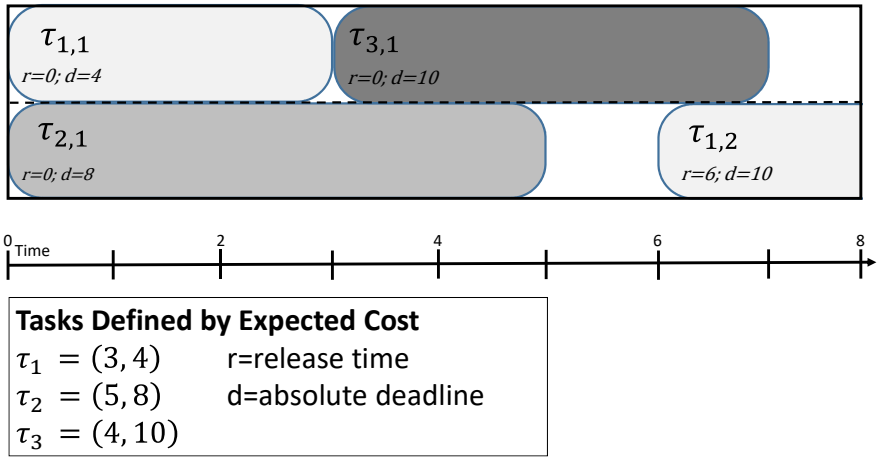

Figure 3.1: Scheduling as if each SMT thread were a separate core.

determining the execution times for each task τ_i given that each job of τ_i is co-scheduled with one or more jobs of any one task τ_j.

Definition 3.1. The costs for tasks τ_i and τ_j when co-scheduled using SMT are given by $C_{i(j)}$ and $C_{j(i)}$, where $C_{i(j)}$ is the cost of a job of τ_i given that the entire job is co-scheduled with one or more jobs of task τ_j and $C_{i(j)}$ is the reverse. For $i = j$, $C_{i(j)}$ is the cost of one job of τ_i given that it is co-scheduled with a second copy of itself. ◄

Because we are using SMT in an SRT system where upper-bounding expected costs is sufficient for bounded tardiness (see Section 2.3.3.5), we are primarily interested in $C_{i(j)}$ and $C_{j(i)}$ as average costs; both values are averages unless stated otherwise.

Example 3.2. Figure 3.2 shows the process used to determine $C_{1(2)}$ and $C_{2(1)}$. One hardware thread continuously runs jobs of τ_1, while the second thread continuously runs jobs of τ_2. Periods are irrelevant; each job begins as soon as the previous job completes. Observe that each job of τ_1 is co-scheduled with different portions of τ_2; the reverse is true as well. In this case, we have

$$C_{1(2)} = \frac{1.0+1.5+1.0+2.0+0.7+1.8}{6} = 1.33 \text{ and } C_{2(1)} = \frac{2.0+1.0+3.0+2.0}{4} = 2.0.$$

◊

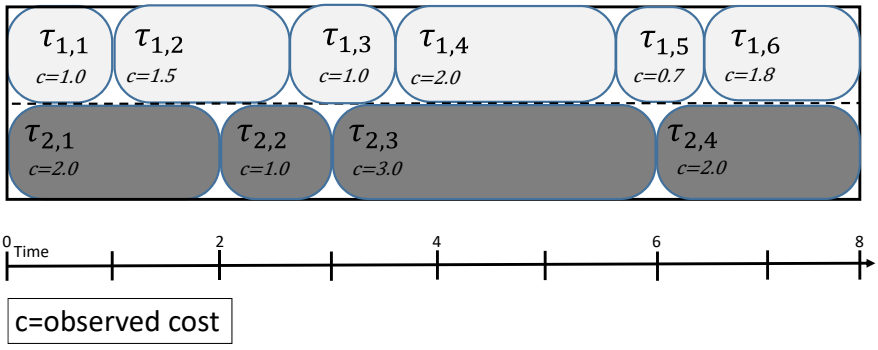

Figure 3.2: High-level depiction of measuring costs for an SRT system.

In a system of n tasks, each task τ_i will have a cost without SMT, C_i, plus a different cost $C_{i(j)}$ for every $j \neq i$. It is not always practical to consider n separate possible costs for every task. In some cases, we make scheduling decisions as if each task τ_i had only two costs, one without SMT, i.e., C_i, and one with SMT.

Definition 3.2. Let τ^h be the set of all tasks with which τ_i can possibly be co-scheduled.[1] The *threaded cost* of τ_i is then given by

$$C_i^h = \max_{\forall\ \tau_j\ \in\ \tau^h\ :\ i \neq j} C_{i(j)}. \tag{3.1}$$

We rely on Assumption 3.1 to justify our use of C_i^h:

Assumption 3.1. The average cost of τ_i given that it can be co-scheduled with any combination of jobs belonging to tasks in set J is at most the greatest value of $C_{i(j)}$ for all j in J. ◄

3.1.3 Experiments: Setup and Execution without SMT

To observe the effects of SMT on execution times, we ran a series of experiments using the TACLeBench sequential benchmarks (Falk et al., 2016), which consist of 23 C implementations of functions commonly found in embedded and real-time systems. We first used this approach in 2019 (Osborne et al., 2019); here we incorporate refinements to the benchmarking process from (Bakita et al., 2021). Details of the benchmark tasks are given in Table 3.2.

To get baseline execution times without SMT, we executed each benchmark 1,000 times. We used the Linux command `chrt` to give benchmarks real-time priority above all normal tasks and the command `taskset` to pin each benchmark to a single core. In addition, we used `isolcpu` to prevent Linux from

[1]We use h rather than t for threaded so as to avoid confusion with t for time.

Table 3.2: Summary of TACLeBench Sequential Benchmarks (Falk et al., 2016)

Name	Description
adpcm_dec	ADPCM decorder
adpcm_enc	ADPCM encoder
ammunition	C compiler arithmetic stress test
anagram	Word anagram computation
audiobeam	Audio beam former
cjpeg_transuff	JPEG image transcoding routines
cjpeg_wrbmp	JPEG image bitmap writing code
dijkstra	All pairs shortest path
epic	Efficient pyramid image coder
fmref	Software FM radio with equalizer
g723_enc	CCITT G.723 encoder
gsm_dec	GSM provisional standard decoder
gsm_enc	GSM provisional standard encoder
h264_dec	H.264 block decoding functions
huff_dec	Huffman decoding with file source to decompress
huff_enc	Huffman encoding with file source to compress
mpeg2	MPEG2 motion estimation
ndes	Complex embedded code
petrinet	Petri net simulation
rijndael_dec	Rijndael AES decryption
rijndael_enc	Rijndael AES encryption
statemate	Statechart simulation of a car window lift control
susan	MR image recognition algorithm

executing any additional processes on the core executing our benchmark and we redirected all interrupt requests (IRQs) to a different core. This setup is the same as what we used in (Bakita et al., 2021). We executed benchmarks one at a time on a dedicated machine; the only other processes running were automatic kernel processes.

For each benchmark, we added a loop enclosing the algorithm's main function. We allowed two iterations, but only recorded the execution time for the second loop iteration. Algorithm 2 shows the structure of our benchmark tasks, including the portion of each task that we timed. The first iteration of the loop brings all data into main memory; the Linux command `mlockall()` ensures that all data will stay in memory. As secondary storage access times can be three to five orders of magnitude greater than main memory access times (Hennessy and Patterson, 2019), locking data in main memory is common practice for real-time systems. At the end of the first loop, we evict all data from the cache to guarantee that the timed data will start with a

cold cache. Using a cold cache adds an additional level of pessimism to our measurements, strengthening Assumption 3.1.

Algorithm 2 Overview of benchmark code.

```
1: mlockall()
2: Execute one-time setup code.
3: for i in {1, 2} do
4:     Begin timer.
5:     Do work.
6:     Stop timer.
7:     Clear cache.
8: end for
9: Record time of second loop only.
```

In practice, real-time programs are frequently structured as endless loops with each job corresponding to one iteration. At the end of each job, the process goes to sleep. This structure is illustrated in Algorithm 3. Unfortunately for us, benchmarks that are widely used in the real-time community are not generally structured as endless or long-running loops. We realize that enclosing an existing benchmark in a loop may cause its execution time to differ from what it would have been without the loop. However, our main interest is in discovering how SMT changes execution times. Any changes to program behavior as a result of the added loop will also apply when SMT is used and will therefore not invalidate our results.

Algorithm 3 Code structure of a real-time task.

```
1: Execute one-time setup code.
2: while True do
3:     Do work.
4:     Sleep until next job.
5: end while
```

Our baseline results are summarized in Table 3.3. To make comparisons between tasks easier, 99th percentiles, maximum observed times, and standard deviations are all given relative to each task's mean. As expected, given that we are repeatedly running identical code, each task shows little variation between its mean and maximum execution times. Benchmarks `petrinet` and `statemate` at first appear to be exceptions—their maximum execution times are 3.79 and 3.89 times their means, respectively. However, both of these benchmarks have extremely short runtimes of a few hundred to a few thousand nanoseconds. For all tasks, absolute differences between mean and maximum runtimes ranged from less than two microseconds (`petrinet`) to almost four hundred microseconds (`dijkstra`).

Table 3.3: Summary of SRT baseline execution times

Benchmark	Mean (ns)	99th percentile/mean	max/mean	Std. Dev./mean
adpcm_dec	84,183	1.18	1.22	0.021
adpcm_enc	84,520	1.19	1.22	0.025
ammunition	19,643,488	1.01	1.01	0.003
anagram	321,219	1.05	1.18	0.015
audiobeam	53,491	1.27	1.44	0.035
cjpeg_transupp	240,596	1.07	1.08	0.019
cjpeg_wrbmp	25,436	1.05	1.72	0.047
dijkstra	6,617,518	1.03	1.06	0.011
epic	342,677	1.05	1.09	0.014
fmref	49,960	1.29	1.38	0.039
g723_enc	63,900	1.23	1.29	0.030
gsm_dec	209,780	1.08	1.09	0.019
gsm_enc	503,814	1.03	1.04	0.011
h264_dec	23,819	1.04	1.93	0.051
huff_dec	37,271	1.08	1.50	0.049
huff_enc	96,053	1.17	1.22	0.026
mpeg2	24,541,106	1.00	1.00	0.001
ndes	11,454	1.11	1.24	0.040
petrinet	619	2.35	3.79	0.325
rijndael_dec	351,366	1.07	1.13	0.016
rijndael_enc	333,479	1.06	1.14	0.016
statemate	5,875	1.15	3.89	0.152
susan	3,157,534	1.01	1.01	0.002

The range of per-task execution times is quite large, running from less than one microsecond for `petrinet` to more than 20 ms for `mpeg2`. This distribution is not ideal for our purposes, but the advantages of using a well-established benchmark set outweigh the disadvantages.

3.1.4 Experiments: Execution Times with SMT

To obtain execution times with SMT, we executed every benchmark alongside every other benchmark. For each pair, we designated a "measured" task and an "interfering" task. The measured task's execution was identical to the baseline setup described in Section 3.1.3. The interfering task was pinned to the second hardware thread of the same core[2] and run in a loop until we had collected 1,000 measurements for the measured task. For the interfering task, we did not clear the cache after each loop nor did we repeatedly start and stop the benchmark as we did for the measured task. If we did so, we would be observing the effects of

[2]In Linux terminology, each thread is a separate processor.

context switches and cache clearing on the measured task rather than the effect of the interfering task. After collecting all measurements for the measured task, we changed the measured and interfering designations; for every task pair τ_i and τ_j, we observed both the execution of τ_i under SMT interference from τ_j and the execution of τ_j under the interference of τ_i. In addition, we observed each τ_i when the interfering task was a second copy of the same task.

To evaluate the effect of SMT on each task, we define a *multithreading score* as the percent increase in τ_i's execution time caused by interference from τ_j.

Definition 3.3. The *SRT multithreading score* $M_{i(j)}$ is defined as

$$M_{i(j)} = \frac{C_{i(j)} - C_i}{C_i}$$

when τ_i is the measured task and τ_j the interfering task. ◀

Example 3.3. Let $C_i = 10$ and $C_{i(j)} = 15$. $M_{i(j)} = 0.5$ holds, i.e., interference from τ_j causes the execution time of τ_i to increase by 50%. ◊

$M_{i(j)} = 0$ would indicate that applying SMT does not increase the execution time of τ_i at all; $M_{i(j)} > 1$ would indicate that τ_i's execution time is more than doubled by SMT. SMT is potentially useful so long as $M_{i(j)} < 1$ holds. There is no expectation that $M_{i(j)} = M_{j(i)}$ will hold.

Theoretically, a negative value for $M_{i(j)}$ implies that using SMT reduces the execution time of τ_i. Our experiments did produce some negative $M_{i(j)}$ values. For all of these values, `petrinet` was the measured task. As seen in Table 3.3, this task has an extremely low baseline execution time and an observed maximum nearly four times its average. For this reason, we pessimistically assume our observed negative $M_{i(j)}$ values reflect unexpectedly high values of C_i rather than indicating that SMT is truly decreasing execution times. Therefore, to avoid adding unwarranted optimism to our results, we report negative values as observed but replace them with 0.01 for all summary statistics and calculations. In future work, we will attempt to determine alternative explanations for negative values. For example, does SMT allow for improvements due to cache pre-fetching? If SMT can be used to reliably reduce execution times, benefits could be significant.

In previous work on SMT and SRT systems (Osborne et al., 2019; Bakita et al., 2021), we reported task costs with SMT divided by task costs without SMT. We use a different metric here for consistency with our reporting on SMT and HRT tasks, discussed in Section 3.3.

Because C_i and $C_{i(j)}$ can be defined as average costs, worst observed costs, or other metrics, $M_{i(j)}$ values are dependent on the exact definitions used for the C values. We consider three different definitions for $M_{i(j)}$: with both C_i and $C_{i(j)}$ defined as (1) mean, (2) 99th percentile, and (3) worst-observed costs.

While we are mainly interested in $C_{i(j)}$ as an average, SMT-aware scheduling algorithms do not have to be based on average costs. Reporting 99th percentile and worst-observed costs provides a starting point for future work. Even when scheduling decisions are based on average costs, knowledge of extreme execution times is helpful; the more a job exceeds its budgeted execution time, the more future jobs of the same task will be delayed. If the effect of SMT on maximum costs were much greater than its effect on average costs, then using SMT might be undesirable even if it appeared useful based on average costs alone. We intend to integrate this consideration into our analysis in future work.

3.1.4.1 Summary Statistics

Summary statistics for all three versions of $M_{i(j)}$ values are given in Table 3.4. In these summary statistics, and the rest of our calculations, negative $M_{i(j)}$ values are replaced with $M_{i(j)} = 0.01$.

We saw mean $M_{i(j)}$ values ranging from 0.29 to 0.34, median $M_{i(j)}$ values from 0.24 to 0.30, and 75th percentile $M_{i(j)}$ values from 0.34 to 0.37. All mean-based values were under 1.0, as were almost all 99th percentile and maximum-based values. These results indicate that given our experimental setup, co-scheduling two tasks similar to our benchmarks will almost always take less time than scheduling them sequentially. This finding is similar to existing results (Bulpin, 2005; Bulpin and Pratt, 2004; Tuck and Tullsen, 2003), but past results have considered only average costs within average conditions.

SMT does not appear to become less effective as we move from comparing mean costs to 99th percentiles and observed maximums. While mean $M_{i(j)}$ is largest when comparing maximum C values and smallest when comparing means, the reverse is true for both the median and 75th percentile of $M_{i(j)}$. If SMT did become less effective as the definition of cost becomes more extreme, we would expect to see $M_{i(j)}$ values increase for all summary statistics, not just the mean, as we changed our definition of $M_{i(j)}$ from being based on averages to 99th percentiles to maximum observed costs.

3.1.4.2 A More Detailed View

In this subsection we consider all $M_{i(j)}$ values, not merely summary statistics.

Table 3.4: Summary statistics for $M_{i(j)}$.

Cost Defs.	mean $M_{i(j)}$	median $M_{i(j)}$	75th per. $M_{i(j)}$	% of $M_{i(j)} < 1$	Std. Dev.
mean	0.29	0.27	0.36	100.00%	0.18
99th per.	0.30	0.30	0.37	99.81%	0.20
max	0.34	0.24	0.34	98.87%	0.65

measured task \ interfering task	adpcm_dec	adpcm_enc	ammunition	anagram	audiobeam	cjpeg_transupp	cjpeg_wrbmp	dijkstra	epic	fmref	g723_enc	gsm_dec	gsm_enc	h264_dec	huff_dec	huff_enc	mpeg2	ndes	petrinet	rijndael_dec	rijndael_enc	statemate	susan	mean (no negatives)
adpcm_dec	0.01	0.02	0.02	0.02	0.02	0.02	0.02	0.03	0.02	0.02	0.02	0.01	0.02	0.01	0.01	0.02	0.02	0.01	0.01	0.02	0.02	0.01	0.01	0.02
adpcm_enc	0.02	0.02	0.02	0.02	0.01	0.02	0.01	0.02	0.02	0.02	0.02	0.02	0.02	0.01	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.01	0.02
ammunition	0.32	0.32	0.56	0.37	0.33	0.36	0.32	0.47	0.37	0.32	0.33	0.33	0.35	0.32	0.32	0.33	0.67	0.32	0.32	0.36	0.37	0.32	0.45	0.37
anagram	0.30	0.30	0.49	0.28	0.30	0.30	0.30	0.53	0.30	0.30	0.30	0.30	0.30	0.30	0.30	0.30	0.58	0.30	0.30	0.30	0.30	0.30	0.29	0.33
audiobeam	0.48	0.48	0.61	0.49	0.46	0.49	0.48	0.87	0.49	0.48	0.49	0.49	0.50	0.48	0.48	0.49	0.72	0.48	0.48	0.49	0.49	0.48	0.48	0.52
cjpeg_transupp	0.35	0.35	0.72	0.35	0.34	0.34	0.35	0.75	0.35	0.35	0.35	0.34	0.35	0.35	0.35	0.35	0.79	0.35	0.34	0.34	0.35	0.35	0.34	0.40
cjpeg_wrbmp	0.13	0.13	0.12	0.14	0.13	0.14	0.13	0.16	0.14	0.13	0.14	0.13	0.14	0.14	0.13	0.14	0.06	0.13	0.13	0.13	0.14	0.13	0.13	0.13
dijkstra	0.12	0.12	0.14	0.10	0.11	0.09	0.07	0.10	0.10	0.16	0.10	0.09	0.10	0.08	0.16	0.09	0.24	0.08	0.09	0.10	0.10	0.09	0.14	0.11
epic	0.31	0.31	0.62	0.31	0.31	0.31	0.31	0.75	0.30	0.31	0.31	0.31	0.31	0.31	0.31	0.31	0.80	0.31	0.31	0.31	0.31	0.31	0.31	0.37
fmref	0.43	0.44	0.49	0.44	0.44	0.44	0.43	0.55	0.43	0.41	0.44	0.43	0.44	0.44	0.44	0.44	0.51	0.44	0.43	0.43	0.44	0.43	0.43	0.45
g723_enc	0.37	0.38	0.79	0.39	0.38	0.37	0.38	0.79	0.38	0.37	0.34	0.37	0.39	0.37	0.37	0.37	0.96	0.37	0.38	0.38	0.37	0.38	0.37	0.44
gsm_dec	0.33	0.33	0.65	0.33	0.33	0.33	0.33	0.65	0.33	0.33	0.33	0.33	0.34	0.33	0.33	0.33	0.81	0.33	0.33	0.33	0.33	0.33	0.33	0.38
gsm_enc	0.17	0.17	0.44	0.17	0.17	0.17	0.17	0.53	0.17	0.17	0.17	0.17	0.15	0.17	0.17	0.17	0.50	0.17	0.17	0.17	0.17	0.17	0.17	0.21
h264_dec	0.20	0.21	0.50	0.20	0.20	0.21	0.20	0.74	0.20	0.21	0.20	0.20	0.21	0.20	0.20	0.20	0.61	0.20	0.21	0.20	0.20	0.21	0.20	0.26
huff_dec	0.21	0.20	0.31	0.22	0.21	0.21	0.21	0.35	0.21	0.21	0.21	0.21	0.22	0.21	0.11	0.22	0.36	0.21	0.21	0.21	0.21	0.21	0.21	0.22
huff_enc	0.21	0.21	0.37	0.22	0.21	0.22	0.21	0.54	0.22	0.21	0.21	0.21	0.22	0.21	0.21	0.12	0.51	0.21	0.20	0.21	0.21	0.21	0.20	0.24
mpeg2	0.27	0.27	0.64	0.30	0.27	0.29	0.26	0.43	0.31	0.26	0.27	0.28	0.31	0.26	0.26	0.27	0.71	0.26	0.26	0.30	0.31	0.26	0.45	0.33
ndes	0.22	0.22	0.35	0.22	0.23	0.22	0.21	0.43	0.22	0.22	0.21	0.22	0.23	0.22	0.22	0.22	0.42	0.16	0.21	0.22	0.21	0.22	0.22	0.24
petrinet	0.20	0.23	0.28	0.19	0.20	0.22	0.24	0.31	0.20	0.19	0.20	0.20	0.24	0.18	0.19	0.19	0.31	0.19	-0.03	0.17	0.13	0.18	0.19	0.21
rijndael_dec	0.11	0.11	0.47	0.11	0.11	0.11	0.11	0.83	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.69	0.11	0.11	0.11	0.11	0.11	0.11	0.19
rijndael_enc	0.16	0.16	0.55	0.16	0.16	0.16	0.16	0.91	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.63	0.16	0.16	0.16	0.16	0.16	0.16	0.23
statemate	0.56	0.58	0.41	0.57	0.57	0.57	0.57	0.83	0.59	0.57	0.58	0.57	0.58	0.58	0.56	0.58	0.26	0.57	0.58	0.56	0.58	0.57	0.57	0.56
susan	0.33	0.33	0.67	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.94	0.33	0.33	0.33	0.33	0.33	0.33	0.37
mean (no negatives)	0.25	0.26	0.44	0.26	0.25	0.26	0.25	0.52	0.26	0.25	0.25	0.25	0.26	0.25	0.25	0.25	0.53	0.25	0.25	0.25	0.25	0.25	0.27	

Figure 3.3: $M_{i(j)}$ values given both C_i and $C_{i(j)}$ are average observed cost.

The full set of $M_{i(j)}$ values are given in Figs. 3.3 through 3.5 . For each $M_{i(j)}$ value task τ_i, i.e., the measured task, is on the left; task τ_j, i.e., the interfering task, is along the top. The right-most column and bottom row give means excluding any negative $M_{i(j)}$ values for each row and column, respectively. The figures are color-coded so that larger values are darker and redder. Color-coding is consistent across all three figures. The relationships between values in the same row or column are arguably more important than exact numerical values. The color coding makes these relationships easier to see. Exact definitions of C_i and $C_{i(j)}$ used to determine the $M_{i(j)}$ values are included in each figure's caption.

For most tasks, increases in execution time when using SMT are fairly consistent regardless of the interfering task. We quantify this observation in Table 3.5, which summarizes the coefficient of variation[3] (C.V.) of $M_{i(j)}$ for each measured task τ_i across all possible interfering tasks τ_j. Visually, this can be seen

[3]The standard deviation divided by the mean.

measured task \ interfering task	adpcm_dec	adpcm_enc	ammunition	anagram	audiobeam	cjpeg_transupp	cjpeg_wrbmp	dijkstra	epic	fmref	g723_enc	gsm_dec	gsm_enc	h264_dec	huff_dec	huff_enc	mpeg2	ndes	petrinet	rijndael_dec	rijndael_enc	statemate	susan	mean (no negatives)
adpcm_dec	0.03	0.03	0.03	0.03	0.03	0.04	0.03	0.02	0.04	0.03	0.04	0.02	0.04	0.02	0.03	0.03	0.01	0.03	0.03	0.03	0.03	0.03	0.02	0.03
adpcm_enc	0.02	0.02	0.02	0.02	0.01	0.03	0.02	0.01	0.02	0.01	0.01	0.01	0.02	0.02	0.02	0.02	0.00	0.02	0.02	0.02	0.02	0.02	0.01	0.02
ammunition	0.34	0.34	0.59	0.38	0.34	0.36	0.34	0.49	0.37	0.34	0.34	0.36	0.40	0.34	0.34	0.34	0.67	0.34	0.34	0.37	0.38	0.34	0.47	0.39
anagram	0.28	0.28	0.55	0.27	0.28	0.28	0.28	0.51	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.56	0.28	0.28	0.28	0.28	0.28	0.28	0.31
audiobeam	0.42	0.41	0.49	0.42	0.40	0.42	0.42	0.74	0.41	0.42	0.42	0.43	0.42	0.42	0.41	0.42	0.60	0.41	0.41	0.42	0.42	0.42	0.39	0.44
cjpeg_transupp	0.33	0.33	0.87	0.33	0.33	0.32	0.33	0.75	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.82	0.33	0.33	0.33	0.33	0.33	0.33	0.39
cjpeg_wrbmp	0.68	0.11	0.12	0.69	0.11	0.71	0.10	0.64	0.73	0.12	0.68	0.14	0.67	0.71	0.12	0.70	0.14	0.12	0.13	0.14	0.70	0.11	0.13	0.37
dijkstra	0.17	0.17	0.21	0.15	0.15	0.16	0.12	0.17	0.15	0.15	0.17	0.15	0.16	0.13	0.14	0.15	0.27	0.14	0.14	0.16	0.15	0.14	0.19	0.16
epic	0.30	0.30	0.74	0.30	0.30	0.30	0.30	0.73	0.29	0.30	0.30	0.30	0.30	0.30	0.30	0.30	0.84	0.30	0.30	0.31	0.30	0.30	0.30	0.36
fmref	0.37	0.37	0.39	0.37	0.37	0.37	0.36	0.44	0.36	0.36	0.37	0.37	0.36	0.37	0.37	0.37	0.42	0.37	0.36	0.37	0.37	0.36	0.36	0.37
g723_enc	0.37	0.37	0.86	0.38	0.37	0.36	0.37	0.75	0.37	0.36	0.34	0.37	0.38	0.37	0.36	0.36	0.84	0.36	0.37	0.37	0.37	0.36	0.37	0.42
gsm_dec	0.31	0.31	0.74	0.31	0.31	0.31	0.31	0.63	0.31	0.31	0.31	0.30	0.31	0.31	0.31	0.31	0.75	0.31	0.31	0.31	0.31	0.31	0.30	0.36
gsm_enc	0.17	0.16	0.56	0.16	0.16	0.16	0.16	0.52	0.16	0.17	0.16	0.16	0.15	0.17	0.16	0.16	0.55	0.17	0.16	0.16	0.16	0.16	0.16	0.21
h264_dec	0.20	0.78	0.64	0.22	0.22	0.77	0.22	0.83	0.20	0.80	0.21	0.20	0.45	0.22	0.21	0.22	0.63	0.77	0.82	0.22	0.22	0.78	0.24	0.44
huff_dec	0.50	0.19	0.40	0.53	0.52	0.52	0.52	0.63	0.53	0.51	0.51	0.51	0.53	0.51	0.44	0.51	0.36	0.49	0.52	0.49	0.53	0.53	0.17	0.48
huff_enc	0.22	0.21	0.37	0.19	0.22	0.22	0.22	0.48	0.22	0.21	0.21	0.22	0.19	0.21	0.22	0.14	0.42	0.21	0.21	0.18	0.18	0.21	0.17	0.23
mpeg2	0.28	0.28	0.66	0.31	0.28	0.30	0.27	0.44	0.33	0.27	0.28	0.30	0.35	0.27	0.27	0.28	0.74	0.27	0.27	0.32	0.32	0.27	0.47	0.34
ndes	0.19	0.19	1.34	0.20	0.20	0.20	0.19	0.41	0.19	0.21	0.19	0.19	0.22	0.19	0.19	0.19	0.35	0.14	0.19	0.20	0.19	0.18	0.19	0.26
petrinet	0.11	0.11	0.01	0.04	0.13	0.12	0.03	0.11	0.04	0.04	0.10	0.04	0.08	0.02	0.05	0.11	0.08	0.03	-0.12	-0.02	-0.01	-0.05	0.06	0.06
rijndael_dec	0.09	0.10	0.62	0.09	0.09	0.11	0.11	0.80	0.09	0.09	0.09	0.09	0.10	0.09	0.09	0.10	0.65	0.11	0.13	0.11	0.13	0.11	0.11	0.18
rijndael_enc	0.15	0.17	0.69	0.16	0.18	0.15	0.17	0.85	0.17	0.15	0.16	0.15	0.16	0.14	0.16	0.14	0.71	0.16	0.14	0.15	0.16	0.15	0.18	0.24
statemate	0.45	0.46	0.43	0.45	0.45	0.45	0.45	0.88	0.47	0.45	0.45	0.45	0.46	0.46	0.46	0.45	0.21	0.45	0.47	0.44	0.45	0.45	0.46	0.46
susan	0.33	0.33	0.77	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.93	0.33	0.33	0.33	0.33	0.33	0.33	0.37
mean (no negatives)	0.27	0.26	0.53	0.28	0.25	0.30	0.25	0.53	0.28	0.27	0.27	0.25	0.29	0.27	0.24	0.27	0.50	0.27	0.27	0.25	0.28	0.27	0.25	

Figure 3.4: $M_{i(j)}$ values given both C_i and $C_{i(j)}$ are 99th percentiles

measured task \ interfering task	adpcm_dec	adpcm_enc	ammunition	anagram	audiobeam	cjpeg_transupp	cjpeg_wrbmp	dijkstra	epic	fmref	g723_enc	gsm_dec	gsm_enc	h264_dec	huff_dec	huff_enc	mpeg2	ndes	petrinet	rijndael_dec	rijndael_enc	statemate	susan	mean (no negatives)
adpcm_dec	0.04	0.01	0.01	0.02	0.03	0.08	0.01	0.05	0.06	0.04	0.02	0.03	0.07	0.07	0.01	0.03	-0.01	0.01	0.03	0.05	0.08	0.01	0.02	0.03
adpcm_enc	0.05	0.07	0.01	0.03	0.00	0.11	0.05	0.00	0.02	0.00	0.04	0.01	0.01	0.05	0.04	0.01	0.00	0.05	0.01	0.03	0.03	0.01	0.01	0.03
ammunition	0.34	0.34	0.59	0.38	0.34	0.36	0.34	0.49	0.38	0.34	0.34	0.36	0.41	0.34	0.34	0.35	0.67	0.34	0.34	0.38	0.38	0.34	0.47	0.39
anagram	0.15	0.15	0.39	0.13	0.17	0.14	0.15	0.35	0.14	0.16	0.15	0.16	0.16	0.16	0.15	0.16	0.40	0.15	0.15	0.16	0.15	0.14	0.15	0.18
audiobeam	0.28	0.26	0.38	0.34	0.26	0.27	0.33	0.59	0.27	0.29	0.28	0.28	0.30	0.29	0.33	0.29	0.49	0.29	0.30	0.28	0.28	0.30	0.27	0.32
cjpeg_transupp	0.34	0.34	0.88	0.36	0.33	0.32	0.33	0.75	0.33	0.35	0.38	0.33	0.35	0.32	0.32	0.34	0.84	0.36	0.34	0.33	0.32	0.33	0.32	0.40
cjpeg_wrbmp	0.11	0.09	0.04	0.09	0.08	0.19	0.14	0.10	0.24	0.09	0.07	0.12	0.23	0.09	0.13	0.11	0.06	0.13	0.29	0.13	0.09	0.11	0.09	0.12
dijkstra	0.15	0.17	0.25	0.15	0.14	0.15	0.10	0.15	0.15	0.16	0.14	0.14	0.16	0.15	0.15	0.14	0.25	0.14	0.14	0.15	0.15	0.14	0.19	0.16
epic	0.27	0.26	0.70	0.27	0.31	0.27	0.27	0.69	0.26	0.29	0.30	0.27	0.27	0.27	0.27	0.27	0.77	0.30	0.27	0.27	0.27	0.30	0.27	0.33
fmref	0.39	0.33	0.35	0.32	0.34	0.38	0.31	0.38	0.32	0.29	0.37	0.31	0.43	0.35	0.33	0.33	0.37	0.36	0.35	0.38	0.32	0.32	0.29	0.34
g723_enc	0.33	0.34	0.81	0.40	0.36	0.38	0.33	0.73	0.32	0.32	0.33	0.36	0.38	0.33	0.34	0.37	0.79	0.35	0.38	0.35	0.41	0.42	0.33	0.41
gsm_dec	0.39	0.31	0.73	0.31	0.30	0.31	0.31	0.63	0.33	0.32	0.32	0.33	0.47	0.32	0.31	0.33	0.74	0.31	0.30	0.31	0.31	0.31	0.30	0.37
gsm_enc	0.17	0.17	0.57	0.16	0.18	0.17	0.16	0.52	0.17	0.18	0.17	0.17	0.18	0.17	0.18	0.17	0.56	0.17	0.16	0.19	0.16	0.18	0.17	0.22
h264_dec	0.01	0.12	0.04	0.03	0.02	0.15	0.02	0.33	0.02	0.13	0.09	0.03	0.00	0.02	0.00	0.18	0.19	0.00	0.04	0.05	0.01	0.03	0.00	0.07
huff_dec	0.15	0.13	0.19	0.14	0.24	0.26	0.14	0.23	0.16	0.22	0.13	0.16	0.16	0.20	0.09	0.18	0.22	0.26	0.21	0.13	0.19	0.16	0.15	0.18
huff_enc	0.17	0.21	0.33	0.18	0.19	0.19	0.22	0.45	0.23	0.19	0.21	0.23	0.16	0.19	0.22	0.11	0.37	0.17	0.19	0.17	0.15	0.22	0.18	0.21
mpeg2	0.28	0.28	0.66	0.31	0.28	0.30	0.27	0.45	0.33	0.27	0.28	0.30	0.35	0.27	0.27	0.28	0.74	0.27	0.28	0.32	0.32	0.27	0.48	0.34
ndes	1.29	1.70	1.24	1.30	1.53	1.20	1.18	1.31	1.27	1.36	1.17	1.24	1.33	1.15	1.24	1.46	1.28	1.38	1.31	1.16	1.28	1.40	1.28	1.31
petrinet	-0.07	6.17	-0.14	-0.01	0.10	-0.11	6.27	-0.06	-0.01	-0.08	-0.07	0.05	-0.09	-0.05	6.26	-0.06	-0.03	7.00	6.15	-0.04	-0.19	-0.17	-0.15	1.61
rijndael_dec	0.06	0.11	0.53	0.10	0.07	0.11	0.11	0.70	0.06	0.09	0.08	0.09	0.07	0.11	0.11	0.10	0.55	0.10	0.11	0.11	0.13	0.09	0.11	0.16
rijndael_enc	0.11	0.15	0.58	0.16	0.11	0.15	0.18	0.76	0.15	0.13	0.12	0.15	0.15	0.10	0.15	0.14	0.59	0.14	0.10	0.12	0.41	0.13	0.16	0.21
statemate	0.11	0.16	0.35	0.08	0.39	0.15	0.10	0.40	0.18	0.09	0.14	0.17	0.73	0.16	0.15	0.51	0.43	0.14	0.14	0.21	0.43	0.13	-0.51	0.23
susan	0.33	0.33	0.77	0.33	0.33	0.37	0.33	0.33	0.33	0.33	0.33	0.33	0.34	0.33	0.33	0.50	0.93	0.33	0.33	0.33	0.33	0.33	0.33	0.38
mean (no negatives)	0.24	0.53	0.45	0.24	0.26	0.26	0.51	0.45	0.25	0.25	0.24	0.24	0.29	0.24	0.51	0.28	0.49	0.55	0.25	0.24	0.27	0.25	0.24	

Figure 3.5: $M_{i(j)}$ values given both C_i and $C_{i(j)}$ are maximum observed values.

Table 3.5: C.V. of $M_{i(j)}$ for each measured task τ_i across all possible interfering tasks τ_j. Values in parentheses exclude interference from the three tasks (`ammunition, dijkstra, mpeg2`) that cause the highest levels of interference.

	mean		99th per.		max	
minimum	0.067	(0.001)	0.053	(0.001)	0.097	(0.038)
median	0.336	(0.025)	0.368	(0.081)	0.426	(0.122)
mean	0.354	(0.065)	0.447	(0.158)	0.554	(0.339)
maximum	1.063	(0.259)	1.145	(0.775)	1.920	(1.760)

Table 3.6: C.V. of $M_{i(j)}$ for each interfering task τ_j across all possible measured tasks τ_i.

	mean	99th per.	max
minimum	0.49	0.46	0.67
median	0.55	0.60	1.06
mean	0.55	0.59	1.34
maximum	0.61	0.67	2.58

in each of Figs. 3.3–3.5 as rows where the color remains largely the same across the entire row; a low C.V. corresponds to a uniformly colored row.

In all three figures, $M_{i(j)}$ values are noticeably higher when τ_j is one of `ammunition`, `dijkstra`, or `mpeg2`. Visually, these higher values result in dark red-orange vertical stripes.. If the interfering effects of these three tasks are not considered, then $M_{i(j)}$ values for each measured task τ_i are even more consistent. The parenthetical values in Table 3.5 show the even smaller C.V. values that result when $M_{i(j)}$ values with j corresponding to `ammunition`, `dijkstra`, or `mpeg2` are excluded.

Without these tasks, the coefficient of variation is less than 0.2 in most cases. Although these three tasks have the highest execution times of our benchmarks (see Table 3.3), recall that interfering tasks are executed as a single endless loop. Per-job execution times are therefore irrelevant. For this reason, we do not suspect a causative relation between execution time and interference.

While timing is largely consistent for τ_i as τ_j changes, the reverse is not true; the degree to which τ_j increases the execution time of τ_i varies dramatically as τ_i changes. Table 3.6 shows for every task τ_j, the coefficient of variation of $M_{i(j)}$ across all possible values of i; no matter what C definitions are used for $M_{i(j)}$ the minimum coefficient of variation is at least 0.49; in all cases, the mean and median coefficients of variation are greater than 0.5.

We were surprised that co-scheduling a task with a second copy of itself does not result in exceptionally poor, i.e., high, $M_{i(j)}$ values. We expected that tasks competing for the same execution units, e.g., floating-point multipliers, would fare poorly with SMT, but this does not seem to be the case. If co-scheduling tasks with themselves did result in poor performance, the main diagonals of Figs. 3.3 through 3.5 would appear as dark red stripes. Because $M_{i(i)}$ is generally close to the average of $M_{i(j)}$ across all values of j, co-scheduling a task with itself may be a reasonable method for a low-effort first-pass evaluation of SMT effectiveness. It may be that if $M_{i(i)}$ is large, then either $M_{i(j)}$ or $M_{j(i)}$ will be large for all j, meaning there would be little benefit to using SMT with τ_i. We plan to investigate this possibility in future work.

3.1.4.3 Key Timing Behaviors Associated with SMT

Before moving on to how we model SMT's effects, we briefly recap four key findings.

First, using SMT given our benchmark tasks, architecture, and execution conditions increases execution times by less than 40% in the majority of cases and by less than 100% in almost all cases. Second, the increase a given task sees to its execution time varies little as the task causing interference changes. Third, execution-time increases caused by each interfering task are not consistent, but vary by measured task. However, a small number of tasks seem to cause significantly greater interference than others. Finally, co-scheduling a task with a second copy of itself *may* be predictive of its overall behavior when using SMT.

3.1.4.4 The Effects of Varying Inputs

In our experiments, every job of each benchmark uses identical hard-coded "input" data. In a real application, inputs will vary across jobs. We have not considered how changing input values may affect execution times either with or without SMT; determining appropriate test inputs for timing analysis is a complex topic even without SMT.

3.1.5 Modeling SMT Timing Behavior: SRT

In this section, we give a process for modeling the timing behavior we have observed. This process can be used to generate systems of synthetic tasks in which cost parameters have relationships that are similar to what we have observed. Our process consists of four steps, detailed below: determining an overall expected value, modeling each task's vulnerability to SMT, modeling the degree to which each task interferes with

others, and finally determining individual $M_{i(j)}$ values as a function of vulnerability, interference, and possibly random variation. This process allows us to separate expected $M_{i(j)}$ values from how those values are related to each other. For example, we could create two systems with the same expected value $M_{i(j)}$ that would allow for different degrees of correlation between values.

3.1.5.1 Overall Expected Value

Our first step is to give an overall expected value μ for all $M_{i(j)}$ values. We use 0.2, 0.4, and 0.6. The first value, 0.2, is optimistic: it is lower than our observed mean when comparing average C_i to average $C_{i(j)}$ values. $\mu = 0.4$ is pessimistic with respect to our observed means, and $\mu = 0.6$ is more pessimistic still. We refer to these values as *optimistic*, *mid-range*, and *pessimistic*.

3.1.5.2 Vulnerability

Based on our observation that $M_{i(j)}$ varies little as j changes, we model each task as having an inherent level of *vulnerability* to SMT.

Definition 3.4. If τ_i is an SRT benchmark task, let its *vulnerability*, V_i, be equal to the observed average of $M_{i(j)}$ across all possible values of j. If τ_i is a synthetic SRT task, let V_i be defined as the expected value of $M_{i(j)}$, i.e.,

$$E[M_{i(j)}] = V_i. \blacktriangleleft \tag{3.2}$$

Allowing $V_i < 0$ could be overly optimistic, as $V_i < 0$ would indicate that the expected result for co-scheduling τ_i was to see τ_i's execution time decrease. For this reason, we select each V_i value from the Exponential distribution with mean μ; Exponential random variables are never negative. We also considered using the Normal and Uniform distributions for V_i. However, using either of those distributions would add unjustified optimism to our model; the Normal distribution can produce negative values, and the Uniform distribution would imply an upper bound on $M_{i(j)}$.

The exponential distribution has the probability density function (pdf) and cumulative distribution functions (cdf) given by

$$f(x,\beta) = \frac{1}{\beta} \cdot e^{-\frac{x}{\beta}} \text{ and}$$

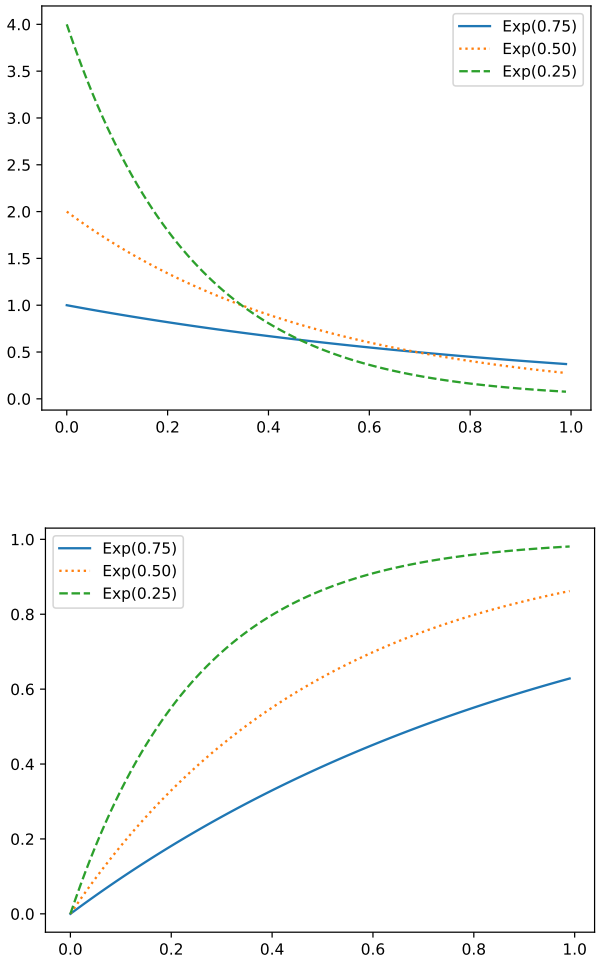

Figure 3.6: Probability density function (pdf, top) and cumulative distribution function (cdf, bottom) for the exponential distributions with expected value $0.25, 0.5$, and 0.75.

$$F(x,\beta) = 1 - e^{\frac{x}{\beta}},$$

respectively; β gives the expected value for the distribution. These functions are shown in Figure 3.6.

The median of the exponential is given by $\mu \ln(2)$. An exponential distribution using $\mu = 0.4$ thus has a median of 0.27, which is equal or close to the medians reported in Table 3.4.

The exponential distribution is not the only possible choice for V_i; so long as all V_i values are selected from distributions with expected value μ, it follows from Expression (3.17) that $E[M_{i(j)}] = \mu$ will hold.

3.1.5.3 Interference

We model interfering tasks as being either *standard* or *harmful*. In our observations, `ammunition`, `dijkstra`, and `mpeg2` would be considered harmful; all other benchmarks would be standard. We define harmful and standard tasks in our model as follows:

Definition 3.5. If τ_j is *harmful*, then

$$E[M_{i(j)}] = a_h \cdot V_i$$

holds for a constant $a_h \geq 1$. Likewise, if τ_j is *standard*, then

$$E[M_{i(j)}] = a_s \cdot V_i$$

holds for a constant $0 < a_s \leq 1$. ◀

We use the binary random variable H_j to denote the event that τ_j is either harmful or standard.

Definition 3.6. Let H_j be a random variable equal to one if τ_j is harmful and 0 otherwise. Furthermore, let h give the probability of a task being harmful, i.e.,

$$\Pr(H_j = 1) = h \text{ and } \Pr(H_j = 0) = (1-h). \blacktriangleleft$$

We can use Definition 3.6 to restate Definition 3.5 in purely mathematical terms:

$$E[M_{i(j)} \mid H_j = 1] = a_h \cdot V_i \tag{3.3}$$

$$E[M_{i(j)} \mid H_j = 0] = a_s \cdot V_i \tag{3.4}$$

Proposition 3.1. In probability, the principle of *linearity of expectations* states that if X can take on values $x_1, x_2, \ldots x_n$ with probabilities $p_1, p_2, \ldots p_n$, then

$$E[X] = \sum_{i=1}^{n} p_i \cdot x_i.$$

We can use Proposition 3.1 to state the expected value of $M_{i(j)}$ in terms of h, a_h, and a_s.

$$E[M_{i(j)}] = h \cdot a_h \cdot V_i + (1-h) \cdot a_s \cdot V_i \tag{3.5}$$

When considering the effect of harmful tasks, it is more intuitive to think of how much worse a harmful task is than a standard task. For this reason, we use

$$r = \frac{a_h}{a_s} \tag{3.6}$$

to describe the effects of a harmful task. Stating both h and r gives all information needed to determine the effects of harmful and standard tasks.

Theorem 3.1. *Given a task system model with interference defined by constants r and $h > 0$ it follows that*

$$a_s = \frac{1}{h \cdot (r-1)+1} \text{ and } a_h = \frac{r}{h \cdot (r-1)+1}.$$

Proof. The result is obtained by algebraically solving Expressions (3.17), (3.5), and (3.6) for a_s and a_h.

Starting from Expression (3.17), we have

$$\begin{aligned}
& E[M_{i(j)}] = V_i \\
\Rightarrow\ & \{\text{By Exp. (3.5)}\} \\
& V_i = h \cdot a_h \cdot V_i + (1-h) \cdot a_s V_i \\
\Rightarrow\ & \\
& 1 = h \cdot a_h + (1-h) \cdot a_s \\
\Rightarrow\ & \{\text{Exp. (3.6) implies } a_h = r \cdot a_s\} \\
& 1 = h \cdot r \cdot a_s + (1-h) a_s \\
\Rightarrow\ & \\
& \frac{1}{h(r-1)+1} = a_s
\end{aligned}$$

The result for a_h follows from Expression (3.6); $a_h = r \cdot a_s$. □

Defining $h = 0$ is equivalent to saying all tasks will be standard; in that case we set $a_s = 1$.

In our experiments, we use $r = 2$ and h values of 0, $\frac{1}{8}$, and $\frac{1}{4}$. $r = 2$ and $h = \frac{1}{8}$ are based on our observations. The remaining h values are included to make our schedulability results more broadly applicable.

3.1.5.4 Determining $M_{i(j)}$

We have shown how to determine $E[M_{i(j)}]$, but not how to actually select $M_{i(j)}$. We consider two basic approaches. In the *fixed* approach, we do not use a probability distribution at all, but set

$$M_{i(j)} = E[M_{i(j)}].$$

This approach models the scenario where $M_{i(j)}$ is entirely determined by the vulnerability of τ_i and the harmfulness of τ_j. This scenario is not very realistic, but it can be useful in making comparisons.

The second approach is to model $M_{i(j)}$ as a random variable with the appropriate expected value. We choose to use an exponential distribution with mean V_i; our reasons are similar to those given in Section 3.1.5.2. We refer to this as the *exponential* approach.

3.2 Timing Analysis for HRT Systems: Is SMT Safe?

"SMT cannot be used for hard real-time because execution times cannot be safely analyzed." This belief is common in the real-time community and something we have heard frequently. In this section, we make the case that if it is possible to analyze execution times to a given standard when not considering SMT, it is reasonable to apply the same standard to execution times with SMT.

Table 3.7 summarizes the notation used in this section and Section 3.3; entries are organized by the order they appear in this section and Section 3.3. The work in this section and Section 3.3 is a continuation of work first done in 2020 (Osborne and Anderson, 2020).

3.2.1 Scheduling Overview and Cost Definition

We start by eliminating as much SMT-induced timing uncertainty as possible. We do so via an approach called *simultaneous co-scheduling.*

Definition 3.7. We say that two jobs are *simultaneously co-scheduled* if both begin execution simultaneously on hardware threads belonging to the same physical core, and when one job completes, the remaining job

Table 3.7: Summary of notation used in Secs. 3.2 and 3.3.

Symbol	Description	Reference	Sections Used
$\tau_{i.a:j.b}$	Simultaneously co-scheduled HRT jobs	Def. 3.7	3.2
$C_{i(j)}^{(1)}$	Cost of HRT task τ_i when simul. co-scheduled with τ_j	Def. 3.9	3.2, 3.3
E_i	Random variable for execution time of a future job of τ_i	Def. 2.13	2.5, 3.2
S_i^q	sWCET; $\Pr(E_i \leq S_i^q) = q$ holds for random variable S_i^q	Def. 3.10	3.2
q	safety level of task τ_i; $\Pr(E_i \leq S_i^q)$	Def. 3.10	3.2
A	Trace: ordered set of consecutive execution times	Def. 2.15	2.5
A_k	kth element of trace A	Def. 2.15	2.5
$A_{\max}$	maximum of trace A	Def. 2.15	2.5
C_i^p	$\Pr(E_i \leq C_i^p) = p$ holds for constant C_i^p	Def. 2.14	2.5
$q_{b(\lvert A\rvert)}$	Lower bound of q given trace of size $\lvert A\rvert$	Def. 3.13	3.2
A^+	Pop. of execution times within experimental world	Def. 3.14	3.2
z	Count of values in A^+ but not A no greater than $A_{\max}$	Def. 3.15	3.2
$q_{c(\lvert A\rvert)}$	Computed value of q given a trace A and population A^+	Def. 3.16	3.2
C_i^p	pWCET of τ_i: $\Pr(E_i \leq C_i^p) = p$	Def. 2.14	2.5, 3.2
f_i	Linear regression function	Def. 3.19	3.3
V_i	Vulnerability of task τ_i	Defs. 3.4 and 3.21	3.1, 3.3

continues on the same core until complete. No other job can be scheduled on the core until both jobs have completed. We use $\tau_{i.a:j.b}$ to denote the simultaneously co-scheduled jobs $\tau_{i.a}$ and $\tau_{j.b}$. ◄

Definition 3.8. We say that two tasks are *simultaneously co-scheduled* if all of their jobs will be simultaneously co-scheduled. ◄

Simultaneous co-scheduling requires additional cost definitions.

Definition 3.9. The cost of τ_i given that it is simultaneously co-scheduled with τ_j is given by $C_{i(j)}^{(1)}$. Likewise, the cost of τ_j given that it is simultaneously co-scheduled with τ_i is given by $C_{j(i)}^{(1)}$. The (1) superscript distinguishes $C_{i(j)}^{(1)}$ from the $C_{i(j)}$ parameter used to describe costs for SRT purposes; the idea behind our notation is that $C_{i(j)}^{(1)}$ gives the cost of τ_i when co-scheduled with one job of τ_j. ◄

Example 3.4. In Fig. 3.7, $C_{1(2)}^{(1)} = 6$ and $C_{2(1)}^{(1)} = 4$. Notice that when $\tau_{2,1}$ completes, $\tau_{1,1}$ continues its execution alone. ◊

We implement simultaneous co-scheduling using barrier synchronization; neither task can proceed until both are ready. When measuring $C_{i(j)}^{(1)}$ (respectively, $C_{j(i)}^{(1)}$), we record the difference between τ_i's (τ_j's) completion time and the starting time of the *first* of the two jobs to begin. In the unlikely event that the job of τ_i begins significantly later than the job of τ_j, the measured $C_{i(j)}^{(1)}$ value will include both the actual execution time of τ_i and the synchronization delay.

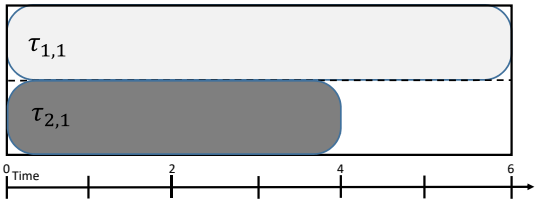

Figure 3.7: Example of simultaneous co-scheduling.

3.2.2 Quantifying Timing Analysis Safety

To show that $C_{i:j}^{(1)}$ can be determined as safely as costs without SMT, we must first define what it means for one cost to be as safe as another. However, our ability to compare costs is limited by our underlying knowledge.

Example 3.5. Consider again rolling a single die. If we know the die has sides numbered one through six, and that each side has equal probability of being rolled, we can say

$$\Pr(\text{randomly selected roll}) \leq 5 = \frac{5}{6}. \tag{3.7}$$

However, this claim relies on our knowledge of the die's physical properties. If our only knowledge of the die is a record of its past rolls—we cannot actually examine the die—we might compute

$$\Pr(\text{randomly selected roll}) \leq \text{maximum observed roll} \tag{3.8}$$

Both quantities—the randomly selected roll and the maximum observed roll—are random variables. $\Diamond$

We define the safety of a given cost similarly to Expression (3.8); we are concerned with the probability that one random variable, a random future execution, is no greater than an observed maximum, which is also a random variable.

Definition 3.10. Given a task τ_i and a random variable E_i corresponding to the execution time of a random future job of τ_i (Definition 2.13), a *safe WCET* (sWCET) S_i^q is a random variable such that

$$\Pr(E_i \leq S_i^q) = q \tag{3.9}$$

holds. We refer to q as the *safety level*[4] of τ_i. ◄

As with the die example, S_i^q is a sample maximum. More specifically, it is the maximum of trace A (Definition 2.15).

Definition 3.11. Given task τ_i and trace A, $S_i^q = A_{max}$. ◄

Definition 3.10 is similar to the definition of a pWCET (Definition 2.14, Section 2.5). The difference is that, S_i^q is a random variable, whereas Definition 2.14 is concerned with the probability of E_i exceeding a constant.

We define the execution-cost parameters C_i, $C_{i(j)}^{(1)}$, and $C_{j(i)}^{(1)}$ in our model so that $C_i = S_i^q$, $C_{i(j)}^{(1)} = S_{i(j)}^q$ and $C_{j(i)}^{(1)} = S_{j(i)}^q$hold for a specified q. The traditional notion of correctness—all jobs are guaranteed to complete on time—cannot be adhered to without known upper bounds on all job execution times. For this reason, we supplement the binary idea of correctness with a quantifiable level of *safety*.

Definition 3.12. Task system τ is q-safe if all tasks and simultaneously co-scheduled task pairs have sWCETs with safety level at least q, and the system would be correctly scheduled if all tasks had true WCETs no greater than the stated sWCETs. ◄

Some may question the wisdom of using sWCETs within a safety-critical context. However, probabilistic reasoning is already present within HRT contexts. In particular, FAA standards for commercial aircraft state acceptable failure rates, essentially giving a probabilistic bound. *We are not weakening HRT correctness; we are making explicit a dependence on timing analysis that is often left implicit.*

Determining an appropriate safety level is an application-specific decision. Our concept of safety is applicable to any system designed around an acceptable failure rate, which could theoretically include applications ranging from streaming media to aviation. We highlight aviation rules to show that probabilistic timing analysis does not preclude systems that have both HRT and safety-critical requirements. However, the number of trials we require increases inversely to the acceptable failure rate, leading to scalability problems for applications with extremely small acceptable failure rates.

3.2.3 Safety Given a True Random Sample

In this subsection, we formally state our assumptions regarding the randomness of our trace A and show what trace size is needed to guarantee a given safety level q given those assumptions.

[4]Note that q is unrelated to safety integrity levels used in risk analysis, despite the similar name.

Assumption 3.2. The following properties hold for E_i—by Definition 2.13, the execution time of a randomly selected future job of τ_i—and all elements A_k of trace A: First, both E_i and all values within A are drawn from the same probability distribution; second, individual A_k values are not dependent on their position within A as denoted by k; and third, individual A_k values are not dependent on other values within A. ◀

If we schedule our system using $A_{max} = S_i^q$ as the task cost, what must q be?

Combining Definition 3.11 and Definition 3.10 gives

$$\Pr(E_i \leq A_{max}) = q. \tag{3.10}$$

We want to determine the value of q. To do so, we need to consider both the relationship between E_i and A_{max} and that between $A_{max} = S_i^q$ and C_i^p for an arbitrary value of p. We make use of three basic rules of probability to determine q, given below. Using these rules, we give a lower bound for q in terms of p and $|A|$ in Lemma 3.1 below.

Proposition 3.2. Let events Y_1 through Y_v partition a probability space, and let X be an event belonging to the same probability space. The law of total probability states that the following property holds: $\Pr(X) = \sum_{i=1}^{v} \Pr(X|Y_i) \cdot \Pr(Y_i)$. ◀

Proposition 3.3. Let X be a possible outcome of a repeated random trial. It follows for a series of trials, where each trial's result is independent from all previous results, that $\Pr(X \text{ holds for some trial}) = 1 - \Pr(X \text{ holds for no trials})$ holds. ◀

Proposition 3.4. Let X be a possible outcome of v repeated, independent trials. Then, $\Pr(X \text{ holds for all trials}) = \Pr(X)^v$. ◀

Lemma 3.1. *Assume Assumptions 3.2 and 3.11 hold. Given a trace A and an arbitrary p associated with some value for C_i^p, the following holds:*

$$q \geq p \cdot \left(1 - p^{|A|}\right). \tag{3.11}$$

Note that since (3.11) holds for an arbitrary p, q is not a function of p.

Proof. The lemma is established by the following derivation:

$$
\begin{aligned}
& q \\
= \; & \{\text{by Exp. (3.10)}\} \\
& \Pr(E_i \leq A_{max}) \\
= \; & \{\text{by Prop. 3.2}\} \\
& \Pr\big(E_i \leq A_{max} | A_{max} \geq C_i^p\big) \cdot \Pr\big(A_{max} \geq C_i^p\big) + \Pr\big(E_i \leq A_{max} | A_{max} < C_i^p\big) \cdot \Pr\big(A_{max} < C_i^p\big) \\
\geq \; & \{\text{since } \Pr\left(E_i \leq A_{max} \mid A_{max} < C_i^p\right) \cdot \Pr\left(A_{max} < C_i^p\right) \geq 0\} \\
& \Pr\left(E_i \leq A_{max} \mid A_{max} \geq C_i^p\right) \cdot \Pr\left(A_{max} \geq C_i^p\right) \\
\geq \; & \{\text{since } \Pr\left(E_i \leq A_{max} \mid A_{max} \geq C_i^p\right) \geq \Pr(E_i \leq C_i^p)\} \\
& \Pr(E_i \leq C_i^p) \cdot \Pr\left(A_{max} \geq C_i^p\right) \\
= \; & \{\text{by Exp. (2.12)}\} \\
& p \cdot \Pr\left(A_{max} \geq C_i^p\right) \\
= \; & \{\text{by the definition of } A_{max} \text{ (Definition 2.15)}\} \\
& p \cdot \Pr(A_k \geq C_i^p \text{ holds for some } A_k \in A) \\
= \; & \{\text{by Prop. 3.3}\} \\
& p \cdot (1 - \Pr(A_k < C_i^p \text{ holds for all } A_k \in A)) \\
= \; & \{\text{by Prop. 3.4}\} \\
& p \cdot (1 - \Pr(A_k < C_i^p)^{|A|}) \\
= \; & \{\text{since } A_k \text{ terms are independent and from the same distribution as } E_i, \text{ per Pre. 3.2}\} \\
& p \cdot (1 - \Pr(E_i < C_i^p)^{|A|}) \\
\geq \; & \{\text{since } \Pr(E_i < C_i^p) \leq \Pr(E_i \leq C_i^p) \; \} \\
& p \cdot (1 - \Pr(E_i \leq C_i^p)^{|A|}) \\
= \; & \{\text{by Exp. (2.12)}\} \\
& p \cdot (1 - p^{|A|}).
\end{aligned}
$$

□

We can now lower-bound q in terms of $|A|$ alone.

Theorem 3.2. *Let Assumptions 3.2 and 3.11 hold. Then it follows that*

$$q \geq \left(\frac{1}{|A|+1}\right)^{\frac{1}{|A|}} \cdot \left(1 - \frac{1}{|A|+1}\right) \tag{3.12}$$

Proof. We first define the lower bound of q given in Expression (3.11) as q^*, i.e., $q \geq q^*$ holds, where

$$q^* = p(1-p^{|A|}).$$

Maximizing q^* will give a lower bound for q in terms of $|A|$ alone. To maximize q^*, we find the value of p for which the first derivative of q^* with respect to p equals 0 and the second derivative is negative. The first derivative is given by

$$\frac{dq^*}{dp} = 1 - (|A|+1) \cdot p^{|A|},$$

and the second by

$$\frac{d^2q^*}{dp^2} = -|A| \cdot (|A|+1) \cdot p^{|A|-1}.$$

Observe that $\frac{d^2q^*}{dp^2} < 0$ holds for all $p > 0$. Consequently, q^* is maximized when $\frac{dq^*}{dp} = 0$.

Solving $\frac{dq^*}{dp} = 0$ for p gives the value of p that maximizes q^*.

$$\begin{aligned}
&\frac{dq^*}{dp} = 1 - (|A|+1) \cdot p^{|A|} \\
&\Rightarrow \\
&0 = 1 - (|A|+1) \cdot p^{|A|} \\
&\Rightarrow \\
&(|A|+1) \cdot p^{|A|} = 1 \\
&\Rightarrow \\
&p^{|A|} = \frac{1}{|A|+1} \\
&p = \left(\frac{1}{|A|+1}\right)^{\frac{1}{|A|}}
\end{aligned}$$

Inserting this value into Expression (3.11) in the place of p gives the result.

$$
\begin{aligned}
& q \geq p\left(1-p^{|A|}\right) \\
\Rightarrow \\
& q \geq \left(\frac{1}{|A|+1}\right)^{\frac{1}{|A|}} \cdot \left(1-\left[\left(\frac{1}{|A|+1}\right)^{\frac{1}{|A|}}\right]^{|A|}\right) \\
\Rightarrow \\
& q \geq \left(\frac{1}{|A|+1}\right)^{\frac{1}{|A|}} \cdot \left(1-\frac{1}{|A|+1}\right)
\end{aligned}
$$

□

Definition 3.13. We refer to the lower bound of Expression (3.12) given trace size $|A|$ as $q_{b(|A|)}$. ◄

For reference, $q_{b(1000)} \approx 0.99212$.

In Section 3.2.5, we will use this result to compare the safety of tasks that do and do not use SMT.

The greatest potential shortfall of this approach is the reliance on Assumption 3.2, which may not hold in practice. However, this obstacle is not unique to us; as mentioned in Section 2.5, EVT methods must also contend with data that may not be as independent as desired. In Secs. 3.2.4 and 3.2.5, we empirically test what happens when Assumption 3.2 does not hold.

3.2.4 Safety Without True Randomness: Obtaining Execution Times

So far, our timing analysis has been theoretical; we have shown that if Assumptions 3.2 and 3.11 hold, then we can safely place a lower bound on the value of q, which we denote as $q_{b(|R|)}$ per Definition 3.13 ("b" denotes our theoretical bound). In practice, however, Assumption 3.2 may not hold. What can we say about timing safety and q in a more realistic setting?

Experimental studies are typically done within a framework that defines an "artificial world," and definitive conclusions can only really be drawn with respect to that "world"—further conclusions concerning the "real" world, though often interesting and relevant, are necessarily speculative. In our context, we need to be able to compare A_{max} values determined from relatively small traces to entire populations. Thus, throughout this section, we assume an artificial world in which only the 23 programs of the TACLeBench

sequential benchmarks (Falk et al., 2016) are of interest. Furthermore, we assume that the entire population of execution times for the task in this world is given by a sequence of 100,000 job executions for each benchmark, which we denote as A^+.

Definition 3.14. Let A^+ be a sequence of 100,000 execution times. Within the artificial world of our experiments, we assume that A^+ gives the entire population of execution times for the task of interest. ◀

In the "real" world, we typically would not have the entire population of possible execution times; if we somehow did, this analysis technique would be unnecessary. With this setup in place, our world thus consists of 23 solo tasks and 529 task pairs, each with 100,000 jobs or job pairs.

As we did for the SRT case in Section 3.1, we executed each benchmark using the Linux command `chrt` to give benchmarks real-time priority, `taskset` to pin each benchmark to a single core, `isolcpu` to prevent the execution of additional processes on the test core, and redirected all IRQs to different CPUs. Also as before, we included an initial untimed loop to bring all data into memory—we use `mlockall` to keep it in memory—and cleared the cache after every job to approximate worst-case running conditions.

Each possible pair of tasks was executed using simultaneous co-scheduling. Within each pair of jobs, the first job to finish would go to sleep until the second job had finished as well. Once both jobs had finished, the cache for both was cleared.

Unlike our SRT experiments, we executed each benchmark and pair of benchmarks as a loop, without stopping execution, until timing data for all jobs had been collected. Algorithm 4 shows this approach.

In the "real" world, appropriately dealing with task inputs in timing analysis is a complex issue. In our setting here, the initial "input" for each benchmark is hard-coded, but the program structure causes the inputs processed by each loop to vary; each job may change the inputs that will be processed by the next job, creating dependencies between sequential execution times.[5]

When we executed benchmarks as we did for the SRT case (Algorithm 2), terminating and restarting the program after every timed loop creates a strong argument that recorded execution times will be independent. In many circumstances, independent execution times are desirable, but if we used fully independent execution times for our HRT benchmark results, we would have no evidence that our results hold for anything other data that fully fits Assumption 3.2. By deliberately using data that violates Assumption 3.2, we show that our

[5]We do not consider whether looping our benchmarks would produce meaningful results; we are using them solely to produce realistic timing data.

Algorithm 4 Overview of HRT benchmark code.

```
1: mlockall()
2: Execute one-time setup code.
3: for i in {1...|A+| + 1} do
4:     Begin timer.
5:     Do work.
6:     Stop timer.
7:     Record time.
8:     Clear cache.
9: end for
```

main result for this section—timing analysis with SMT is not inherently inferior to timing analysis without SMT—holds within an experimental world that departs from the assumption of perfect independence.

3.2.5 Safety Without True Randomness: Analyzing Execution Times

In this subsection, we compare the ability of actual 1,000-job traces to predict the execution time behavior of 100,000 job "populations" to the prediction computed from Expression (3.12) with $|A| = 1,000$ that a future execution time has probability at least 0.99212 of being no greater than the maximum of a 1,000-job trace.

We denote the number of "future" jobs—execution times in A^+ but not in A—no greater than $A_{\max}$ as z.

Definition 3.15. Let z be the number of values in A^+ but not A, i.e., A^+ with the first $|A|$ values excluded, and no value greater than $A_{\max}$.

To avoid overloading q, we define a term for the value we actually compute given a trace and population.

Definition 3.16. Given a trace size $|A|$ and a population A^+, and assuming that A consists of the first $|A|$ execution times in A^+, let the result of computing q per Expression (3.10) be denoted $q_{c(|A|)}$ ("c" denotes a computed value of q). ◄

We can calculate $q_{c(|A|)}$ as the number of future jobs no greater than $A_{\max}$ divided by the total number of future jobs. In other words,

$$q_{c(|A|)} = \frac{z}{|A^+| - |A|} \tag{3.13}$$

We compare $q_{c(|A|)}$ and $q_{b(|A|)}$ to assess our ability to safely schedule a task system. When

$$q_{c(|A|)} \geq q_{b(|A|)} \tag{3.14}$$

holds for a given task then we know, at least within the artificial world of our experiments, that scheduling a system on the basis of observed maximums is $q_{b(|A|)}$ safe (Definition 3.12).

Example 3.6. Let A consist of the five execution times $(3,3,4,8,6)$. Per Definition 3.13, $q_{b(5)}$ is given by

$$\begin{aligned}
&\left(\frac{1}{|A|+1}\right)^{\frac{1}{|A|}} \cdot \left(1-\frac{1}{|A|+1}\right) \\
&= \left(\frac{1}{6}\right)^{\frac{1}{5}} \cdot \left(1-\frac{1}{6}\right) \\
&\approx 0.63.
\end{aligned}$$

This result tells us that if Assumption 3.2 holds, then the probability q that a future value drawn from the same population will be no greater than the sample maximum, eight, is at least 0.63.

Now suppose we have knowledge of the population from which A is drawn: $A^+ = (3,3,4,8,6,1,6,10,4,4)$.[6] It follows that $z = 4$; A^+ excluding A has 4 values no greater than 5. We have

$$\begin{aligned}
&q_{c(5)} \\
&= \frac{z}{|A^+| - |A|} \\
&= \frac{4}{5}.
\end{aligned}$$

Because $q_{c(5)} \geq q_{b(5)}$ holds ($\frac{4}{5} \geq 0.63$), we conclude that using $A_{\max}$ as an sWCET with $q = 0.63$ is safe.

$\Diamond$

We can compute $q_{c(|A|)}$ only within artificial worlds where we can view the entire population A^+. Within such an artificial world, we ask whether tasks using SMT can be as safe as tasks not using SMT.

3.2.6 Safety Without True Randomness: Results

When we calculated $q_{c(1000)}$ for each of our 23 solo tasks, we found a minimum $q_{c(1000)}$ of 0.995, a mean of 0.999, and a maximum of 1.0. We conclude from $q_{c(1000)} > 0.992$ holding that these tasks, which are part of our artificial world, are safe. This conclusion holds even without Assumption 3.2. Table 3.8 shows all $q_{c(1000)}$ values along with additional summary statistics.

[6] We ignore for the moment that A^+ has, by definition, 100,000 elements.

Table 3.8: Summary of HRT baseline execution times.

Benchmark	First 1,000 jobs mean (ns)	First 1,000 jobs max (ns)	All 100,000 jobs mean (ns)	All 100,000 jobs max (ns)	$q_{c(1000)}$
adpcm_dec	84,666	145,183	84,583	145,398	0.99999
adpcm_enc	84,620	103,613	84,678	113,240	0.99876
ammunition	19,629,643	19,766,332	19,642,379	234,097,200	0.99886
anagram	310,793	331,264	309,347	529,686	0.99508
audiobeam	52,056	74,748	51,993	80,338	0.99987
cjpeg_transupp	240,626	265,175	239,256	271,531	0.99981
cjpeg_wrbmp	25,467	46,021	25,530	52,447	0.99959
dijkstra	6,580,290	6,789,719	6,583,733	220,840,048	0.99734
epic	334,410	356,238	334,066	368,017	0.99981
fmref	52,864	111,731	43,178	111,731	1.00000
g723_enc	62,569	86,057	62,632	94,841	0.99981
gsm_dec	208,810	230,989	209,386	269,162	0.99965
gsm_enc	490,436	510,441	490,001	737,621	0.99770
h264_dec	23,096	48,088	23,146	105,186	0.99990
huff_dec	29,096	47,322	29,126	59,734	0.99774
huff_enc	79,499	97,636	79,421	107,618	0.99398
mpeg2	24,519,668	24,565,208	24,546,819	614,412,800	0.99939
ndes	10,467	15,355	10,610	38,213	0.99873
petrinet	164	821	223	18,951	0.99983
rijndael_dec	352,363	373,130	352,365	404,731	0.99913
rijndael_enc	332,262	351,646	332,287	406,740	0.99587
statemate	5,883	33,254	5,887	33,254	1.00000
susan	3,144,910	3,160,378	3,151,315	217,072,704	0.99973

Now we change one element of our world: we allow SMT. Are the tasks still safe?

For our 529 task pairs using SMT, the mean and maximum $q_{c(1000)}$ across all pairs—values for $C_{i(j)}$ and $C_{j(i)}$ are calculated separately—were found to be 0.999 and 1.0, respectively. Additional results, saved as a CSV file, are included in our online appendix (`https://www.cs.unc.edu/~shosborn/dissertation/`).

More importantly than mean and maximum values, the minimum $q_{c(1000)}$ value was 0.993, which is greater than our threshold of 0.992. The answer to the question of safety is yes: when we allow SMT into our world, execution times remain safe.

3.2.7 Rules for Using SMT

Based on our comparisons of $q_{c(1000)}$ and $q_{b(1000)}$ with and without SMT, we propose the following rules for applying SMT to HRT scheduling.

1. SMT shall only be used via means of simultaneous co-scheduling.

2. If $|A|$ execution time samples are considered safe to upper-bound execution times without SMT, then $|A|$ samples shall be considered safe to upper-bound execution times with SMT.

When these rules are followed, simultaneously co-scheduled task pairs can be part of a $q_{b(|A|)}$-safe system, at least within our experimental world. At first glance, this result may seem rather weak; our artificial world is not the real world. However, scheduling research is most often done in an artificial world where execution times are known. Our results indicate that an artificial world where SMT is safe is no less reasonable than the artificial worlds already used in research.

3.3 HRT Benchmark Results: Characterizing Execution Times

Our next task is to examine how SMT affects the execution times of simultaneously co-scheduled tasks relative to their execution times without SMT. In this section, we give our observations regarding execution time and develop a method for modeling SMT's effects in an HRT context. Continuing from our rules for using SMT in Section 3.2.7, we define C_i, C_j, and $C_{i(j)}^{(1)}$ for each task and pair of tasks in our benchmark data as the maximum of each sample.

3.3.1 The HRT Multithreading Score

Definition 3.17. Let the *HRT multithreading score* $M_{i(j)}^{(1)}$ denote the increase in τ_i's execution time when co-scheduled with τ_j relative to the minimum of C_i and C_j, e.g.,

$$M_{i(j)}^{(1)} = \frac{C_{i(j)}^{(1)} - C_i}{\min(C_i, C_j)} \text{ or,} \tag{3.15}$$

$$C_{i(j)}^{(1)} = C_i + M_{i(j)}^{(1)} \cdot \min(C_i, C_j). \tag{3.16}$$

The exact interpretation of $M_{i(j)}^{(1)}$ depends on the relative values of C_i and C_j.

If $C_j \leq C_i$ holds, then $M_{i(j)}^{(1)}$ give the cost to τ_i of being co-scheduled with τ_j per unit of τ_j's execution time. If $C_i \geq C_j$ holds, then $M_{i(j)}^{(1)}$ is equivalent to the SRT metric of $M_{i(j)}$ (Def. 3.3); it gives the percent increase to τ_i's execution time given that τ_i is co-scheduled with τ_j.

If $C_i \geq C_j$ and $M_{i(j)}^{(1)} \geq 1$ both hold, then there is no benefit to pairing τ_i and τ_j together. If $M_{i(j)}^{(1)} < 1$ holds, then pairing jobs of the two tasks is potentially beneficial, with lower values indicating greater benefit.

If $M_{i(j)}^{(1)} < 0$ holds, then $\tau_{i:j}$ actually requires less measured time to execute than τ_i alone. If $C_i < C_j$ holds, then co-scheduling the tasks may still be beneficial even if $M_{i(j)}^{(1)} \geq 1$ so long as $M_{j(i)}^{(1)} \leq 1$ holds.

Example 3.7. Let τ_1 and τ_2 be such that $C_1 = 5$, $C_2 = 20$, $C_{1(2)}^{(1)} = 12$, and $C_{2(1)}^{(1)} = 22$. It follows that

$$M_{1(2)}^{(1)} = \frac{12-5}{5} = \frac{7}{5} \text{ and}$$
$$M_{2(1)}^{(1)} = \frac{22-20}{5} = \frac{2}{5}.$$

Using SMT saves time; the co-scheduled tasks require 22 time units, whereas 25 units would be required to schedule the tasks sequentially. However, looking at $M_{1(2)}^{(1)}$ could incorrectly give the impresssion that SMT is not helpful. ◊

Fig. 3.8 shows $M_{i(j)}^{(1)}$ calculated for all pairs of tasks. The figure is organized with tasks ordered from top to bottom and left to right by decreasing baseline execution time; $C_i > C_j$ holds for all pairs above and to the right of the main diagonal, with C_j decreasing as we move to the right and C_i decreasing as we move down. Within each row, $\frac{C_i}{C_j}$ is minimized on the left and maximized on the right, with the main diagonal having $\frac{C_i}{C_j} = 1$. Fig. 3.9 shows only the main diagonal and lower half of Fig. 3.8, i.e., $M_{i(j)}^{(1)}$ values for which $C_i \leq C_j$ holds. Fig. 3.10 shows only $M_{i(j)}^{(1)}$ values for which $C_j \leq C_i$ holds. As with the SRT case, the figures are color-coded so that greater $M_{i(j)}^{(1)}$ values are darker and redder. Color-coding is consistent across all figures.

When $C_i \leq C_j$ holds, in Fig. 3.9 and the bottom half of Fig. 3.8, $M_{i(j)}^{(1)}$ appears uncorrelated with $\frac{C_i}{C_j}$. Visually, rows do not show a left-to-right color gradation. From the perspective of τ_i, it makes no difference whether τ_j completes at the same time as τ_i or much later.

When $C_i \geq C_j$ holds, in Fig. 3.10 and the top half of Fig. 3.8, then a positive correlation between $M_{i(j)}^{(1)}$ and $\frac{C_i}{C_j}$ is obvious; rows get redder and darker to the right. This correlation indicates that combining tasks tends to become less efficient as the difference between their costs increases.

Example 3.8. Let τ_1, τ_2, and τ_3 be such that $C_1 = 20$, $C_2 = 10$, and $C_3 = 5$. If $C_{1(2)}^{(1)}$ and $C_{1(3)}^{(1)}$ are both equal to 25, then $M_{1(2)}^{(1)} = \frac{5}{10} = 0.5$ and $M_{1(3)}^{(1)} = \frac{5}{5} = 1$. While co-scheduling τ_1 and τ_2 reduces the total time required to schedule both tasks, co-scheduling τ_1 and τ_3 does not do so. ◊

measured task \ interfering task	mpeg2	ammunition	dijkstra	susan	gsm_enc	rijndael_dec	epic	rijndael_enc	anagram	cjpeg_transupp	gsm_dec	adpcm_dec	fmref	adpcm_enc	huff_enc	g723_enc	audiobeam	h264_dec	huff_dec	cjpeg_wrbmp	statemate	ndes	petrinet
mpeg2	0.94	0.76	0.51	0.86	0.74	0.76	5.13	0.80	0.83	0.74	0.73	0.43	0.66	14.64	1.08	0.94	1.01	0.74	1.16	0.99	0.97	1.99	35.62
ammunition	0.70	0.66	0.42	0.68	0.20	0.17	6.15	-0.08	-0.01	-0.41	-0.30	-1.30	-1.79	5.51	-1.86	18.19	-2.53	-5.84	-4.01	-6.76	-5.59	-6.88	-188.80
dijkstra	0.25	0.27	0.25	0.36	0.37	2.48	2.40	-0.08	3.14	3.92	4.43	7.05	6.85	0.16	0.57	-0.31	10.44	1.08	17.63	16.33	1.00	65.77	924.16
susan	0.95	0.78	0.87	0.96	4.77	0.81	0.93	6.85	7.21	9.10	10.45	16.07	0.57	0.58	24.66	0.90	0.83	0.65	49.88	50.68	51.15	8.07	2893.55
gsm_enc	0.57	0.58	0.56	0.55	0.62	0.60	0.74	0.63	0.79	0.92	1.06	1.51	2.08	2.13	2.41	2.61	2.95	4.52	4.91	4.86	6.41	0.73	262.34
rijndael_dec	0.67	0.61	0.83	0.64	0.68	0.73	0.66	0.72	0.64	0.57	0.79	1.12	0.32	0.42	0.51	1.72	2.17	3.33	3.29	0.29	0.10	10.25	186.88
epic	0.91	0.81	0.77	0.87	0.91	0.81	0.88	0.79	0.77	0.74	0.67	0.32	0.47	0.44	0.68	0.74	3.20	0.49	4.78	4.89	6.98	15.52	287.38
rijndael_enc	0.72	0.72	0.88	0.69	0.71	0.77	0.69	0.76	0.69	0.79	0.90	1.27	1.53	1.64	1.87	2.05	2.37	0.43	3.43	3.72	5.12	11.40	213.03
anagram	0.67	0.67	0.60	0.68	0.66	0.63	0.66	0.66	0.69	0.64	0.81	0.29	1.46	1.64	1.81	1.97	0.55	3.36	3.51	3.36	4.77	0.36	185.67
cjpeg_transupp	0.55	0.57	0.56	0.58	0.58	0.55	0.63	0.61	0.61	0.64	0.51	0.60	0.82	0.80	0.32	1.25	1.35	1.91	2.27	-0.29	2.77	6.35	122.26
gsm_dec	0.38	0.41	0.35	0.43	0.38	0.37	0.42	0.41	0.40	0.43	0.41	0.46	0.16	0.58	0.56	0.70	0.14	-0.27	1.12	1.15	1.70	3.63	68.84
adpcm_dec	-0.26	-0.25	-0.26	-0.26	-0.26	-0.27	-0.26	-0.27	-0.23	-0.26	-0.23	-0.27	-0.34	-0.38	-0.40	-0.43	-0.51	-0.94	-0.94	-0.84	-1.33	-2.95	-55.67
fmref	0.31	0.31	0.30	0.32	0.33	0.27	0.30	0.26	0.32	0.29	0.30	0.28	0.25	0.31	0.33	0.34	0.36	0.30	0.36	0.43	0.23	0.64	3.46
adpcm_enc	0.10	0.04	0.05	0.03	0.06	0.01	0.03	0.04	0.04	0.04	0.03	0.03	0.05	0.05	0.03	0.05	0.08	-0.03	0.06	0.17	-0.12	-0.22	-5.90
huff_enc	0.67	0.63	0.72	0.47	0.66	0.47	0.55	0.48	0.62	0.51	0.53	0.35	0.39	0.36	0.46	0.54	0.52	0.66	0.49	0.38	0.91	1.54	36.98
g723_enc	0.57	0.68	0.50	0.63	0.62	0.52	0.68	0.60	0.65	0.65	0.67	0.45	0.46	0.47	0.63	0.64	0.54	0.43	0.71	0.37	0.12	1.72	-4.62
audiobeam	0.52	0.57	0.65	0.53	0.50	0.46	0.49	0.46	0.52	0.54	0.47	0.58	0.41	0.53	0.52	0.46	0.52	0.29	0.29	0.22	0.01	0.68	19.96
h264_dec	0.04	0.12	0.24	0.08	0.13	-0.28	-0.10	0.03	0.22	-0.02	-0.03	0.07	-0.08	0.04	-0.04	-0.03	0.11	0.01	-0.01	-0.03	-0.12	-0.21	-22.16
huff_dec	0.64	0.48	0.48	0.31	0.47	0.22	0.15	0.32	0.37	0.36	0.33	0.23	0.41	0.25	0.62	0.42	0.28	0.33	0.28	0.27	0.09	0.31	1.86
cjpeg_wrbmp	0.06	0.26	0.09	0.14	0.06	-0.05	0.01	0.06	0.10	0.17	0.09	0.02	0.08	0.10	0.12	0.04	0.12	-0.06	0.00	0.18	-0.03	-0.10	-3.75
statemate	-0.22	-0.17	-0.27	-0.30	-0.46	-0.32	-0.13	-0.28	-0.34	0.00	-0.32	-0.73	-0.71	-0.73	-0.14	-0.48	-0.25	-0.23	-0.29	-0.20	-0.62	-0.62	-14.82
ndes	1.12	0.97	0.09	0.98	0.07	1.30	1.02	1.53	1.05	0.98	1.42	0.95	0.86	1.08	0.02	0.94	1.04	0.13	0.62	0.61	0.69	0.91	12.05
petrinet	0.70	1.07	10.28	1.07	0.73	2.56	0.91	10.82	1.06	0.83	0.92	1.00	1.15	1.18	0.64	0.86	0.80	0.57	3.57	11.53	0.85	0.67	0.38

Figure 3.8: $M^{(1)}_{i(j)}$ values for all pairs.

measured task \ interfering task	mpeg2	ammunition	dijkstra	susan	gsm_enc	rijndael_dec	epic	rijndael_enc	anagram	cjpeg_transupp	gsm_dec	adpcm_dec	fmref	adpcm_enc	huff_enc	g723_enc	audiobeam	h264_dec	huff_dec	cjpeg_wrbmp	statemate	ndes	petrinet
mpeg2	0.94	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
ammunition	0.70	0.66	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
dijkstra	0.25	0.27	0.25	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
susan	0.95	0.78	0.87	0.96	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
gsm_enc	0.57	0.58	0.56	0.55	0.62	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
rijndael_dec	0.67	0.61	0.83	0.64	0.68	0.73	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
epic	0.91	0.81	0.77	0.87	0.91	0.81	0.88	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
rijndael_enc	0.72	0.72	0.88	0.69	0.71	0.77	0.69	0.76	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
anagram	0.67	0.67	0.60	0.68	0.66	0.63	0.66	0.66	0.69	x	x	x	x	x	x	x	x	x	x	x	x	x	x
cjpeg_transupp	0.55	0.57	0.56	0.58	0.58	0.55	0.63	0.61	0.61	0.64	x	x	x	x	x	x	x	x	x	x	x	x	x
gsm_dec	0.38	0.41	0.35	0.43	0.38	0.37	0.42	0.41	0.40	0.43	0.41	x	x	x	x	x	x	x	x	x	x	x	x
adpcm_dec	-0.26	-0.25	-0.26	-0.26	-0.26	-0.27	-0.26	-0.27	-0.23	-0.26	-0.23	-0.27	x	x	x	x	x	x	x	x	x	x	x
fmref	0.31	0.31	0.30	0.32	0.33	0.27	0.30	0.26	0.32	0.29	0.30	0.28	0.25	x	x	x	x	x	x	x	x	x	x
adpcm_enc	0.10	0.04	0.05	0.03	0.06	0.01	0.03	0.04	0.04	0.04	0.03	0.03	0.05	0.05	x	x	x	x	x	x	x	x	x
huff_enc	0.67	0.63	0.72	0.47	0.66	0.47	0.55	0.48	0.62	0.51	0.53	0.35	0.39	0.36	0.46	x	x	x	x	x	x	x	x
g723_enc	0.57	0.68	0.50	0.63	0.62	0.52	0.68	0.60	0.65	0.65	0.67	0.45	0.46	0.47	0.63	0.64	x	x	x	x	x	x	x
audiobeam	0.52	0.57	0.65	0.53	0.50	0.46	0.49	0.46	0.52	0.54	0.47	0.58	0.41	0.53	0.52	0.46	0.52	x	x	x	x	x	x
h264_dec	0.04	0.12	0.24	0.08	0.13	-0.28	-0.10	0.03	0.22	-0.02	-0.03	0.07	-0.08	0.04	-0.04	-0.03	0.11	0.01	x	x	x	x	x
huff_dec	0.64	0.48	0.48	0.31	0.47	0.22	0.15	0.32	0.37	0.36	0.33	0.23	0.41	0.25	0.62	0.42	0.28	0.33	0.28	x	x	x	x
cjpeg_wrbmp	0.06	0.26	0.09	0.14	0.06	-0.05	0.01	0.06	0.10	0.17	0.09	0.02	0.08	0.10	0.12	0.04	0.12	-0.06	0.00	0.18	x	x	x
statemate	-0.22	-0.17	-0.27	-0.30	-0.46	-0.32	-0.13	-0.28	-0.34	0.00	-0.32	-0.73	-0.71	-0.73	-0.14	-0.48	-0.25	-0.23	-0.29	-0.20	-0.62	x	x
ndes	1.12	0.97	0.09	0.98	0.07	1.30	1.02	1.53	1.05	0.98	1.42	0.95	0.86	1.08	0.02	0.94	1.04	0.13	0.62	0.61	0.69	0.91	x
petrinet	0.70	1.07	10.28	1.07	0.73	2.56	0.91	10.82	1.06	0.83	0.92	1.00	1.15	1.18	0.64	0.86	0.80	0.57	3.57	11.53	0.85	0.67	0.38

Figure 3.9: $M^{(1)}_{i(j)}$ values where $C_j \geq C_i$ holds.

measured task \ interfering task	mpeg2	ammunition	dijkstra	susan	gsm_enc	rijndael_dec	epic	rijndael_enc	anagram	cjpeg_transupp	gsm_dec	adpcm_dec	fmref	adpcm_enc	huff_enc	g723_enc	audiobeam	h264_dec	huff_dec	cjpeg_wrbmp	statemate	ndes	petrinet
mpeg2	0.94	0.76	0.51	0.86	0.74	0.76	5.13	0.80	0.83	0.74	0.73	0.43	0.66	14.64	1.08	0.94	1.01	0.74	1.16	0.99	0.97	1.99	35.62
ammunition	x	0.66	0.42	0.68	0.20	0.17	6.15	-0.08	-0.01	-0.41	-0.30	-1.30	-1.79	5.51	-1.86	18.19	-2.53	-5.84	-4.01	-6.76	-5.59	-6.88	-188.80
dijkstra	x	x	0.25	0.36	0.37	2.48	2.40	-0.08	3.14	3.92	4.43	7.05	6.85	0.16	0.57	-0.31	10.44	1.08	17.63	16.33	1.00	65.77	924.16
susan	x	x	x	0.96	4.77	0.81	0.93	6.85	7.21	9.10	10.45	16.07	0.57	0.58	24.66	0.90	0.83	0.65	49.88	50.68	51.15	8.07	2893.55
gsm_enc	x	x	x	x	0.62	0.60	0.74	0.63	0.79	0.92	1.06	1.51	2.08	2.13	2.41	2.61	2.95	4.52	4.91	4.86	6.41	0.73	262.34
rijndael_dec	x	x	x	x	x	0.73	0.66	0.72	0.64	0.57	0.79	1.12	0.32	0.42	0.51	1.72	2.17	3.33	3.29	0.29	0.10	10.25	186.88
epic	x	x	x	x	x	x	0.88	0.79	0.77	0.74	0.67	0.32	0.47	0.44	0.68	0.74	3.20	0.49	4.78	4.89	6.98	15.52	287.38
rijndael_enc	x	x	x	x	x	x	x	0.76	0.69	0.79	0.90	1.27	1.53	1.64	1.87	2.05	2.37	0.43	3.43	3.72	5.12	11.40	213.03
anagram	x	x	x	x	x	x	x	x	0.69	0.64	0.81	0.29	1.46	1.64	1.81	1.97	0.55	3.36	3.51	3.36	4.77	0.36	185.67
cjpeg_transupp	x	x	x	x	x	x	x	x	x	0.64	0.51	0.60	0.82	0.80	0.32	1.25	1.35	1.91	2.27	-0.29	2.77	6.35	122.26
gsm_dec	x	x	x	x	x	x	x	x	x	x	0.41	0.46	0.16	0.58	0.56	0.70	0.14	-0.27	1.12	1.15	1.70	3.63	68.84
adpcm_dec	x	x	x	x	x	x	x	x	x	x	x	-0.27	-0.34	-0.38	-0.40	-0.43	-0.51	-0.94	-0.94	-0.84	-1.33	-2.95	-55.67
fmref	x	x	x	x	x	x	x	x	x	x	x	x	0.25	0.31	0.33	0.34	0.36	0.30	0.36	0.43	0.23	0.64	3.46
adpcm_enc	x	x	x	x	x	x	x	x	x	x	x	x	x	0.05	0.03	0.05	0.08	-0.03	0.06	0.17	-0.12	-0.22	-5.90
huff_enc	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0.46	0.54	0.52	0.66	0.49	0.38	0.91	1.54	36.98
g723_enc	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0.64	0.54	0.43	0.71	0.37	0.12	1.72	-4.62
audiobeam	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0.52	0.29	0.29	0.22	0.01	0.68	19.96
h264_dec	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0.01	-0.01	-0.03	-0.12	-0.21	-22.16
huff_dec	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0.28	0.27	0.09	0.31	1.86
cjpeg_wrbmp	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0.18	-0.03	-0.10	-3.75
statemate	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	-0.62	-0.62	-14.82
ndes	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0.91	12.05
petrinet	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0.38

Figure 3.10: $M_{i(j)}^{(1)}$ values where $C_j \leq C_i$ holds.

3.3.2 $M_{i(j)}^{(1)}$ as a Function of $\frac{C_i}{C_j}$

To better understand the relationship between $M_{i(j)}^{(1)}$ and $\frac{C_i}{C_j}$, we plotted $M_{i(j)}^{(1)}$ as a function of $\frac{C_i}{C_j}$. To account for the correlation not existing when $\frac{C_i}{C_j} \leq 1$ holds, we define the *adjusted cost ratio.*

Definition 3.18. Let the *adjusted cost ratio of τ_i and τ_j be defined as*

$$\max\left(\frac{C_i}{C_j}, 1\right).$$

For each τ_i, we only consider τ_j when $\frac{C_i}{C_j} \leq 10$ holds. This restriction—10 is a somewhat arbitrary threshold—allows us to focus on the most relevant cases, as we do not expect co-scheduling tasks with dramatically different baseline execution times to be beneficial. Fig. 3.11 shows $M_{i(j)}^{(1)}$ values for all task pairs where $\frac{C_i}{C_j} \leq 10$ holds.

In our summary statistics and analysis, we replace negative $M_{i(j)}^{(1)}$ values with 0.01. Our reasons for doing so are the same as for the SRT case discussed in Section 3.1.4.

Graphs of $M_{i(j)}^{(1)}$ as a function of adjusted cost ratio are shown in Figs. 3.12 through 3.15, organized alphabetically by benchmark name. We do not include graphs for `ndes` or `petrinet`; for those benchmarks,

measured task \ interfering task	mpeg2	ammunition	dijkstra	susan	gsm_enc	rijndael_dec	epic	rijndael_enc	anagram	cjpeg_transupp	gsm_dec	adpcm_dec	fmref	adpcm_enc	huff_enc	g723_enc	audiobeam	h264_dec	huff_dec	cjpeg_wrbmp	statemate	ndes	petrinet
mpeg2	0.94	0.76	0.51	0.86	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
ammunition	0.70	0.66	0.42	0.68	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
dijkstra	0.25	0.27	0.25	0.36	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
susan	0.95	0.78	0.87	0.96	4.77	0.81	0.93	6.85	7.21	x	x	x	x	x	x	x	x	x	x	x	x	x	x
gsm_enc	0.57	0.58	0.56	0.55	0.62	0.60	0.74	0.63	0.79	0.92	1.06	1.51	2.08	2.13	2.41	2.61	2.95	x	x	x	x	x	x
rijndael_dec	0.67	0.61	0.83	0.64	0.68	0.73	0.66	0.72	0.64	0.57	0.79	1.12	0.32	0.42	0.51	1.72	2.17	3.33	3.29	0.29	x	x	x
epic	0.91	0.81	0.77	0.87	0.91	0.81	0.88	0.79	0.77	0.74	0.67	0.32	0.47	0.44	0.68	0.74	3.20	0.49	4.78	4.89	x	x	x
rijndael_enc	0.72	0.72	0.88	0.69	0.71	0.77	0.69	0.76	0.69	0.79	0.90	1.27	1.53	1.64	1.87	2.05	2.37	0.43	3.43	3.72	x	x	x
anagram	0.67	0.67	0.60	0.68	0.66	0.63	0.66	0.66	0.69	0.64	0.81	0.29	1.46	1.64	1.81	1.97	0.55	3.36	3.51	3.36	4.77	x	x
cjpeg_transupp	0.55	0.57	0.56	0.58	0.58	0.55	0.63	0.61	0.61	0.64	0.51	0.60	0.82	0.80	0.32	1.25	1.35	1.91	2.27	-0.29	2.77	x	x
gsm_dec	0.38	0.41	0.35	0.43	0.38	0.37	0.42	0.41	0.40	0.43	0.41	0.46	0.16	0.58	0.56	0.70	0.14	-0.27	1.12	1.15	1.70	x	x
adpcm_dec	-0.26	-0.25	-0.26	-0.26	-0.26	-0.27	-0.26	-0.27	-0.23	-0.26	-0.23	-0.27	-0.34	-0.38	-0.40	-0.43	-0.51	-0.94	-0.94	-0.84	-1.33	-2.95	x
fmref	0.31	0.31	0.30	0.32	0.33	0.27	0.30	0.26	0.32	0.29	0.30	0.28	0.25	0.31	0.33	0.34	0.36	0.30	0.36	0.43	0.23	0.64	x
adpcm_enc	0.10	0.04	0.05	0.03	0.06	0.01	0.03	0.04	0.04	0.04	0.03	0.03	0.05	0.05	0.03	0.05	0.08	-0.03	0.06	0.17	-0.12	-0.22	x
huff_enc	0.67	0.63	0.72	0.47	0.66	0.47	0.55	0.48	0.62	0.51	0.53	0.35	0.39	0.36	0.46	0.54	0.52	0.66	0.49	0.38	0.91	1.54	x
g723_enc	0.57	0.68	0.50	0.63	0.62	0.52	0.68	0.60	0.65	0.65	0.67	0.45	0.46	0.47	0.63	0.64	0.54	0.43	0.71	0.37	0.12	1.72	x
audiobeam	0.52	0.57	0.65	0.53	0.50	0.46	0.49	0.46	0.52	0.54	0.47	0.58	0.41	0.53	0.52	0.46	0.52	0.29	0.29	0.22	0.01	0.68	x
h264_dec	0.04	0.12	0.24	0.08	0.13	-0.28	-0.10	0.03	0.22	-0.02	-0.03	0.07	-0.08	0.04	-0.04	-0.03	0.11	0.01	-0.01	-0.03	-0.12	-0.21	x
huff_dec	0.64	0.48	0.48	0.31	0.47	0.22	0.15	0.32	0.37	0.36	0.33	0.23	0.41	0.25	0.62	0.42	0.28	0.33	0.28	0.27	0.09	0.31	x
cjpeg_wrbmp	0.06	0.26	0.09	0.14	0.06	-0.05	0.01	0.06	0.10	0.17	0.09	0.02	0.08	0.10	0.12	0.04	0.12	-0.06	0.00	0.18	-0.03	-0.10	x
statemate	-0.22	-0.17	-0.27	-0.30	-0.46	-0.32	-0.13	-0.28	-0.34	0.00	-0.32	-0.73	-0.71	-0.73	-0.14	-0.48	-0.25	-0.23	-0.29	-0.20	-0.62	-0.62	x
ndes	1.12	0.97	0.09	0.98	0.07	1.30	1.02	1.53	1.05	0.98	1.42	0.95	0.86	1.08	0.02	0.94	1.04	0.13	0.62	0.61	0.69	0.91	x
petrinet	0.70	1.07	10.28	1.07	0.73	2.56	0.91	10.82	1.06	0.83	0.92	1.00	1.15	1.18	0.64	0.86	0.80	0.57	3.57	11.53	0.85	0.67	0.38

Figure 3.11: $M^{(1)}_{i(j)}$ values where $\frac{C_i}{C_j} \leq 10$ holds.

no $\frac{C_i}{C_j}$ values fall between 1 and 10, making any calculated regression function meaningless. Due to the wide range of possible values, each graph has a different scale. All graphs are summarized in Table 3.9. For each graph, we include a linear function $f_i\left(\max(\frac{C_i}{C_j},1)\right)$ to quantify the relationship between $\frac{C_i}{C_j}$ and $M^{(1)}_{i(j)}$.

Definition 3.19. Let $f_i\left(\max(\frac{C_i}{C_j},1)\right)$ be a linear function that approximates $M^{(1)}_{i(j)}$ as a function of $\max(\frac{C_i}{C_j},1)$ using the least-squares method.

In regression analysis, the *residual* for a given data point (x_ℓ, y_ℓ), where x is the dependent variable and y the independent variable, is defined as $y_\ell - f(x_\ell)$, i.e., the difference between the actual value y_ℓ and the predicted value $f(x_\ell)$. The least-squares method finds the linear function f that minimizes the sum of the residuals squared.

R^2 gives the proportion of total variation in dependent values that is explained by variation in independent values. In our case, $R^2 = 1$ would mean that $M^{(1)}_{i(j)}$ is perfectly modeled as a linear function of $\max(\frac{C_i}{C_j},1)$. $R^2 = 0.75$ implies that variation in $\max(\frac{C_i}{C_j},1)$ accounts for 75% of variation in $M^{(1)}_{i(j)}$.

Definition 3.20. Given a set of (x_1,y_1), (x_2,y_2),...(x_n,y_n) and a linear function $f(x)$,

$$R^2 = \frac{\sum_{\forall \ell}(y_\ell - f(x_\ell))^2}{\sum_{\forall \ell}(y_\ell - \bar{y})^2}. \blacktriangleleft$$

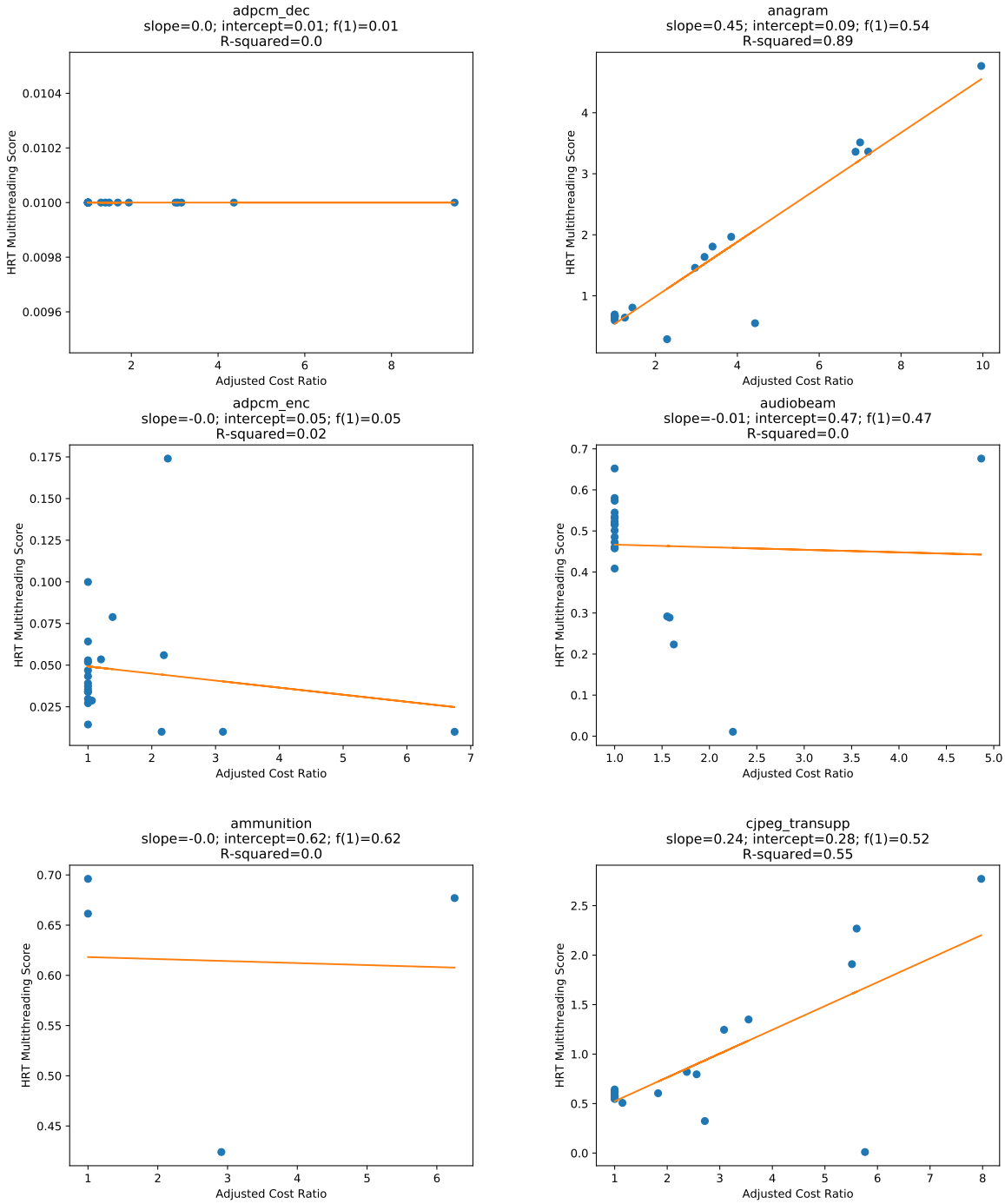

Figure 3.12: $M^{(1)}_{i(j)}$ as a function of max $\left(\frac{C_i}{C_j}, 1\right)$ for benchmarks `adpcm_dec`, `adpcm_enc`, `ammunition`,`anagram`, `audiobeam`, and `cjpeg_transupp`.

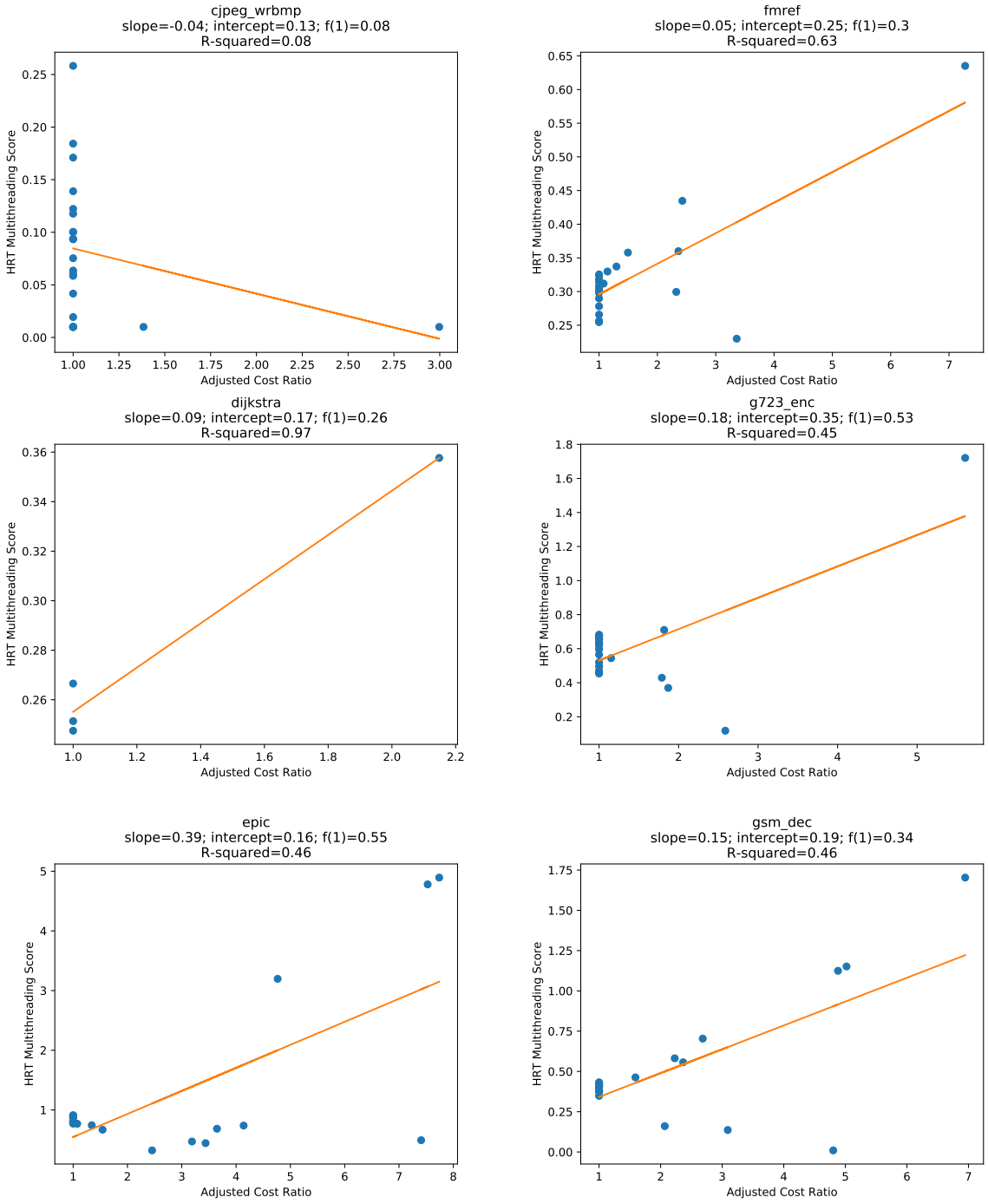

Figure 3.13: $M_{i(j)}^{(1)}$ as a function of $\max\left(\frac{C_i}{C_j}, 1\right)$ for benchmarks `cjpeg_wrbmp`, `dijkstra`, `epic`,`fmref`, `g723_enc`, and `gsm_dec`.

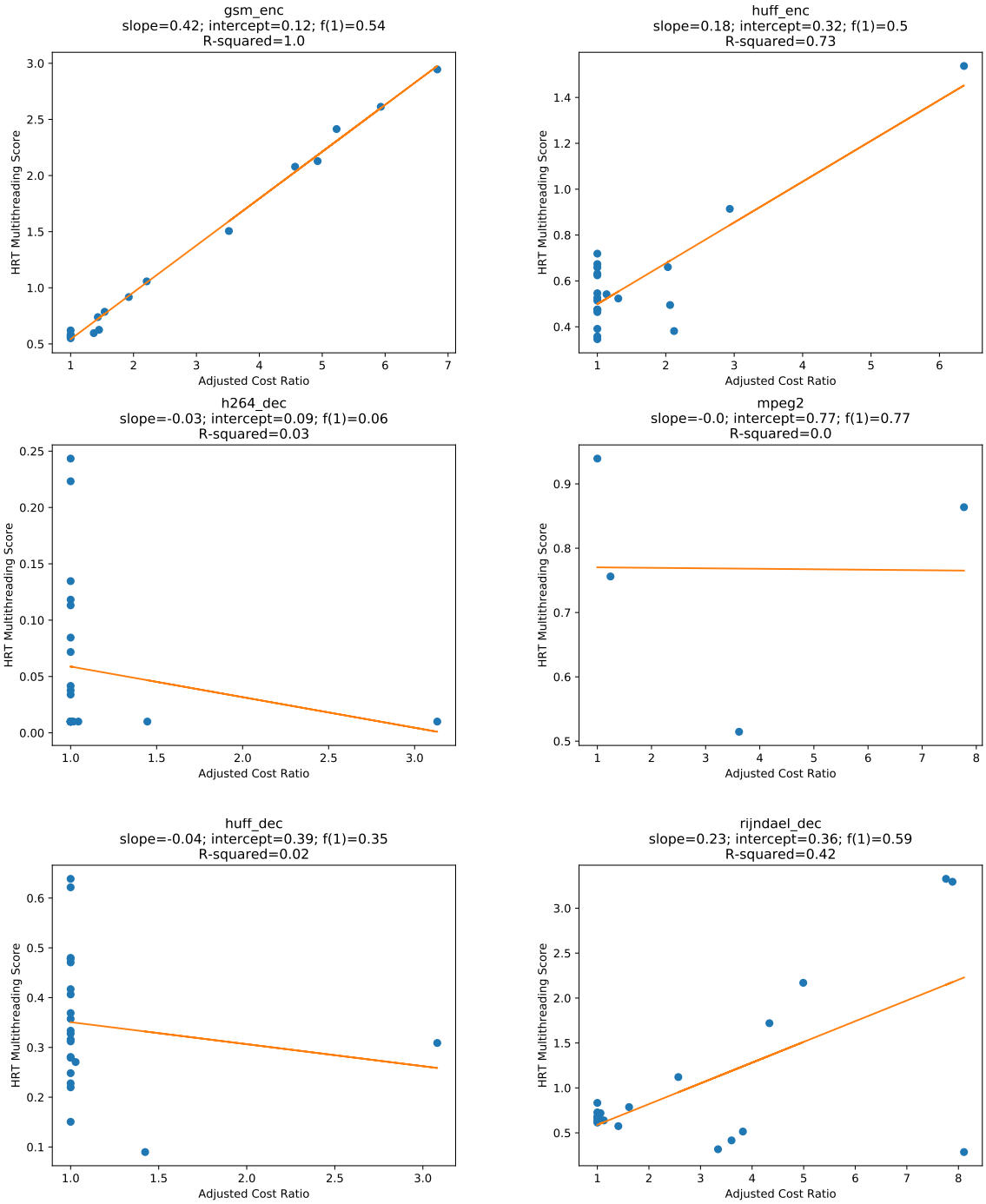

Figure 3.14: $M^{(1)}_{i(j)}$ as a function of max $\left(\frac{C_i}{C_j}, 1\right)$ for benchmarks `gsm_enc`, `h264_dec`, `huff_dec`,`huff_enc`, `mpeg2`, and `rijndael_dec`.

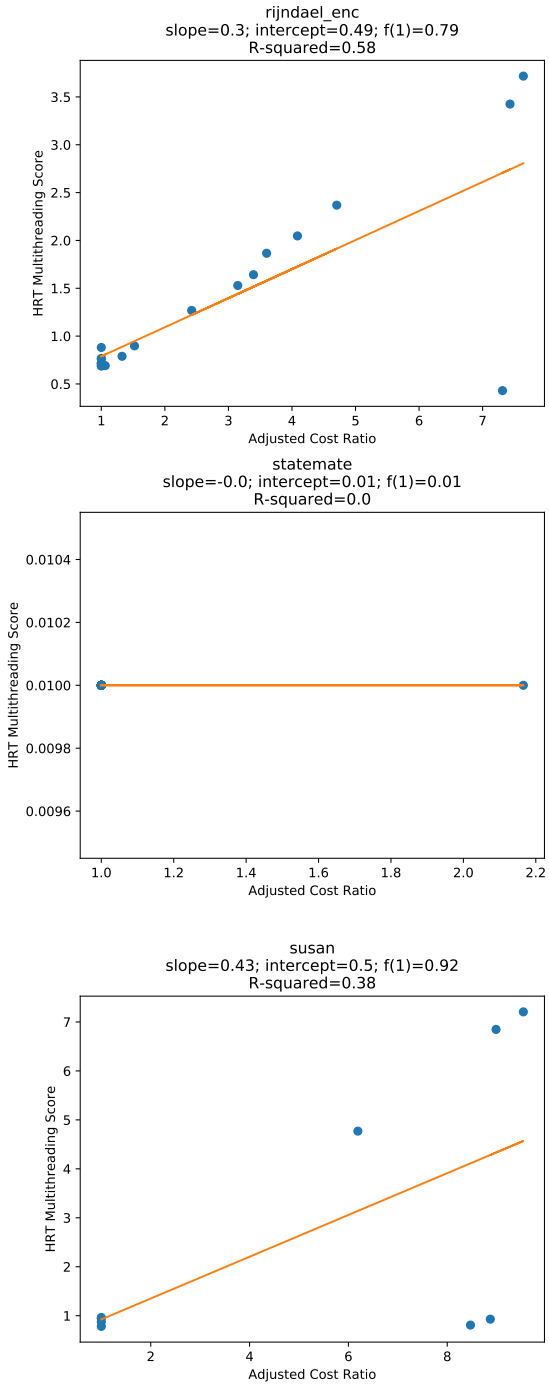

Figure 3.15: $M_{i(j)}^{(1)}$ as a function of $\max\left(\frac{C_i}{C_j}, 1\right)$ for benchmarks `rijndael_enc`, `statemate`, and `susan`.

Table 3.9: Summary of $M_{i(j)}^{(1)}$ as a function of $\frac{C_i}{C_j}$

	Summary of linear regression $f\left(\max\left(\frac{C_i}{C_j},1\right)\right)$				Summary for $\frac{C_i}{C_j}$	
task	slope	intercept	R^2	$f_i(1)$	mean	C.V.
adpcm_dec	-0.200	0.010	0.000	0.010	0.010	0.000
adpcm_enc	-0.004	0.053	0.023	0.049	0.044	0.467
ammunition	-0.002	0.620	0.001	0.618	0.679	0.036
anagram	0.447	0.094	0.894	0.542	0.659	0.044
audiobeam	-0.006	0.473	0.001	0.467	0.514	0.111
cjpeg_transupp	0.240	0.283	0.546	0.524	0.588	0.056
cjpeg_wrbmp	-0.043	0.128	0.076	0.085	0.087	0.763
dijkstra	0.089	0.166	0.975	0.255	0.255	0.039
epic	0.386	0.160	0.458	0.546	0.849	0.065
fmref	0.045	0.251	0.626	0.296	0.295	0.083
g723_enc	0.184	0.347	0.447	0.531	0.588	0.139
gsm_dec	0.148	0.194	0.464	0.342	0.398	0.064
gsm_enc	0.418	0.125	0.995	0.542	0.578	0.046
h264_dec	-0.027	0.086	0.031	0.059	0.066	1.130
huff_dec	-0.044	0.395	0.022	0.351	0.365	0.356
huff_enc	0.178	0.321	0.734	0.499	0.525	0.221
mpeg2	-0.001	0.771	0.000	0.770	0.939	NA
ndes	NA	NA	NA	0.010	0.835	0.510
petrinet	NA	NA	NA	0.010	2.355	1.464
rijndael_dec	0.231	0.359	0.415	0.589	0.694	0.113
rijndael_enc	0.304	0.485	0.580	0.789	0.742	0.085
statemate	0.000	0.010	0.000	0.010	0.010	0.000
susan	0.427	0.497	0.381	0.924	0.892	0.093
minimum	-0.200	0.010	0.000	0.010	0.010	0.000
median	0.089	0.251	0.415	0.467	0.578	0.089
mean	0.132	0.277	0.365	0.383	0.564	0.268
maximum	0.447	0.771	0.995	0.924	2.355	1.464

The title of each graph includes the line's intercept, slope, the line's value at

$$\max\left(\frac{C_i}{C_j},1\right)=1,$$

e.g., $f_i(1)$, and R^2 value. These values, along with additional summary data, are also included in Table 3.9.

Of our 21 graphs, 12 had a positive slope, with values ranging from 0.447 (`anagram`) to 0.045 (`fmref`). Among the positively sloped graphs, the mean slope was 0.258 and the median slope was 0.226 (mean of `cjpeg_transupp` and `rijndael_dec`). R^2 values for the positively sloped graphs ranged from a maximum of 0.995 (`gsm_enc`) to a minimum of 0.381 (`susan`), with a mean of 0.626 and a median of 0.563.

Of the 11 graphs with zero or negative slopes, `adpcm_dec` had a zero slope due to negative $M_{i(j)}^{(1)}$ values being set to 0.01. The minimum slope was -0.04 (`huff_dec`). The mean was -0.012 and the median -0.002 (`ammunition`). R^2 values for negatively sloped graphs were all close to zero, with a maximum of 0.076, a mean of 0.014, and a median of 0.001.

Based on these observations, particularly the small magnitude of negative slopes, we conclude that modeling $M_{i(j)}^{(1)}$ as a positively-sloped linear function of the adjusted cost ratio for τ_i and τ_j is a reasonable choice. We discuss the construction of f_i in the next section.

3.3.3 Modeling SMT Timing Behavior: HRT

In this subsection, we describe, in the context of producing synthetic task systems, how we determine $M_{i(j)}^{(1)}$, and thereby $C_{i(j)}^{(1)}$, given C_i and C_j.

As before, we assign each task a vulnerability value V_i. The definition of V_i (Def. 3.4) is expanded to include HRT tasks.

Definition 3.21. If τ_i is an HRT benchmark task, let its *vulnerability*, V_i, be equal to the observed average of $M_{i(j)}^{(1)}$ across all values of j for which $C_i \leq C_j$ holds. If τ_i is a synthetic HRT task, let V_i be defined as the expected value of $M_{i(j)}^{(1)}$ given that $C_i \leq C_j$ holds, i.e.,

$$E\left[M_{i(j)}^{(1)} \mid C_i \leq C_j\right] = V_i. \blacktriangleleft \tag{3.17}$$

In terms of the linear regression function f_i, V_i gives the functions value when $\frac{C_i}{C_j} \leq 1$ holds, i.e.,

$$f_i(1) = V_i. \tag{3.18}$$

As in the SRT case, we choose V_i from an exponential distribution. We allow the optimistic, mid-range, and pessimistic means, respectively, of 0.35, 0.55, and 0.75. The mid-range mean is slightly less than the mean of $M_{i(j)}^{(1)}$ when $\frac{C_i}{C_j} \leq 1$ holds, as seen in the right-side portion of Table 3.9.

We justify our continued use of vulnerability by the generally low C.V. values in Table 3.9. These values correspond to the rows of Fig. 3.9, which shows only task pairs where $\frac{C_i}{C_j} \leq 1$ holds, having little variation in color. We do not see evidence for harmfulness as we did for the SRT case. For this reaon, we do not model harmfulness for HRT tasks.

For the slope of f_i, we allow three possible values: 0, 0.15, and 0.30. The middle value, 0.15, is slightly greater than the average of our observed slopes. Within each synthetic task system, all f_i functions will have the same slope. This approach is not very realistic, but it will allow us to easily evaluate the impact of SMT becoming less effective as $\frac{C_i}{C_j}$ increases on schedulability.

With a slope and the value of f_i defined for $f_i(1)$, we can easily calculate $f(i)$ for any value of $\frac{C_i}{C_j}$. However, if $\frac{C_i}{C_j} > 10$ holds, then we do not allow τ_i and τ_j to be co-scheduled and thus have no need to model $M_{i(j)}^{(1)}$. We use f_i to determine the expected value of $M_{i(j)}^{(1)}$, i.e.,

$$E\left[M_{i(j)}^{(1)}\right] = f_i\left(\max\left(\frac{C_i}{C_j}, 1\right)\right) \tag{3.19}$$

Finally, we determine $M_{i(j)}^{(1)}$. In the *fixed* approach, we again allow no additional randomization; we simply have

$$M_{i(j)}^{(1)} = f_i\left(\max\left(\frac{C_i}{C_j}, 1\right)\right).$$

Alternatively, we again use the *exponential* approach where $M_{i(j)}^{(1)}$ is drawn from an exponential function with a mean equal to $f_i(\frac{C_i}{C_j})$.

3.4 Conclusions and Future Work

In this chapter, we have evaluated the timing effects of SMT in both SRT and HRT contexts. We measured the execution times of benchmark tasks with and without SMT enabled, with the SMT-enabled case allowing for interference from all other tasks using SMT. Our data suggest SMT has benefits in both SRT and HRT contexts; we explore these benefits in detail in the remaining chapters.

In Sec. 3.2, we have shown that when SMT-enabled jobs are scheduled with care, producing a safe, measurement-based timing analysis is comparable in difficulty to doing the same for jobs not using SMT. We have also highlighted a subtlety of measurement-based timing analysis—any estimate based on random measured data is itself a random variable—that is not always emphasized, and that could compromise safety, if not accounted for. In Sec. 3.3, we gave our observations on SMT's effects on execution times, including the importance of the relative execution times of the two tasks involved. Based on these observations, we gave a model for creating plausible synthetic execution times.

In the future, we want to look more deeply in the negative multithreading scores we observed in both the SRT and HRT cases. If we determine that negative values are more than statistical noise, we will develop scheduling algorithms to exploit this behavior. In addition, we aim to expand the measurement-based analysis of this chapter into a code-based analysis, conduct micro-benchmark experiments to learn more about SMT implementation details, take a deeper look at the statistical aspects of sound timing analysis methods and modeling, and investigate the effects of changing job inputs on SMT's timing effects.

CHAPTER 4: SCHEDULING SOFT REAL-TIME SYSTEMS WITH SMT

In this chapter, we discuss our work on enabling SMT-aware scheduling for SRT systems. We divide the overall problem of using SMT to support SRT systems into two sub-problems. First, in Section 4.1, we show how to schedule a system of tasks where some, but not all, tasks use SMT. Second, in Section 4.2 we decide which tasks should actually use SMT. The difficulty here is that how using SMT will affect any given task is a function of what other tasks are using SMT, creating a circular problem. Finally, in Section 4.3, we evaluate our methods by testing their ability to schedule millions of synthetic task systems. The work in this chapter is based on (Osborne et al., 2019). Notation for this chapter is summarized in Tables 4.1 and 4.2. Terms are organized by the order they appear within this chapter.

4.1 Scheduling With and Without SMT

In this section, we assume a task system has already been split into two sub-systems: a set of tasks that will be using SMT, and a set of tasks that will not be using SMT. Based on such a split system, we give a mathematical test to determine if scheduling the overall system on a given hardware platform is possible.

Definition 4.1. *Physical tasks* are tasks that are only allowed to execute without SMT, i.e., using the full physical core. Subsystem τ^p is the set of all physical tasks in τ. *Threaded tasks* are tasks that are allowed to use SMT. Subsystem τ^h is the set of all threaded tasks in τ. The number of tasks per subsystem is denoted by n^p and n^h, respectively. ◄

Physical tasks have cost C_i and utilization $u_i^p = u_i$ exactly as they would without SMT. We omit the superscript p when the meaning is unambiguous. Threaded tasks have cost

$$C_i^h = \max_{\forall\ \tau_j\ \in\ \tau^h\ :\ i \neq j} C_{i(j)}$$

(the expression duplicates Exp. (3.1); we repeat it for convenience) and utilizations

$$u_{i(j)} = \frac{C_{i(j)}}{T_i} \text{ and} \tag{4.1}$$

Table 4.1: Summary of notation used in Chapter 4.

Symbol	Description	Reference	Sections Used
τ^p	Tasks that do not use SMT	Def. 4.1	4.1, 4.2
τ^h	Tasks that do use SMT	Def. 4.1	4.1, 4.2
$C_{i(j)}$	Cost of an SRT task τ_i when co-scheduled with τ_j	Def. 3.1	3.1, 4.1
C_i^h	$\max_{\forall\ \tau_j\ \in\ \tau^h\ :\ i\neq j} C_{i(j)}$.	Def. 3.2	3.1, 4.1
$u_{i(j)}$	util. of an SRT task τ_i when co-scheduled with τ_j	Exp. 4.1	4.1, 4.2
u_i^h	$\max_{\forall\ \tau_j\ \in\ \tau^h\ :\ i\neq j} u_{i(j)}$.	Exp. 4.2	4.1, 4.2
n^h, n^p	Counts of tasks using/ not using SMT.	Def. 4.1	4.1
U^h	Total util. of all tasks using SMT	Def. 4.2	4.1
U^p	Total util. of all tasks not using SMT.	Def. 4.2	4.1
U^E	Effective Utilization: $U^p + \frac{U^h}{2}$	Def. 4.2	4.1
π^h	Sub-platform scheduling only tasks using SMT.	Def. 4.3	4.1
π^p	Sub-platform scheduling only tasks not using SMT.	Def. 4.3	4.1
m^h, m^p	Number of whole cores for tasks with/ without SMT	Def. 4.3	4.1
b^h	Proportion of shared core for tasks using SMT.	Def. 4.3	4.1
b^p	Proportion of shared core for tasks not using SMT.	Def. 4.3	4.1
$M_{i(j)}$	SRT Multithreading Score; $\frac{C_{i(j)}-C_i}{C_i}$	Def. 3.3	3.1, 4.3
V_i	Vulnerability of task τ_i	Defs. 3.4 and 3.21	3.1, 3.3, 4.3
h	Probability of a task being harmful	Def. 3.6	3.1, 4.3
RSA	Relative Schedulable Area	Def. 4.7	4.3

Table 4.2: Notation originally from (Devi and Anderson, 2006).

Symbol	Description	Reference	Section Used
τ_H, τ_L	Sets of all "high" and "low" tasks	Def. 4.5	4.1
$u_{\max}(\tau_L)$	Util. of highest-util task in τ_L.	Def. 4.5	4.1
U_{sum}	Total util. of both τ_H and τ_L	Def. 4.5	4.1
U_L	Sum of the $\min(\lceil U_{sum}\rceil - 2, n)$ largest util. tasks in τ_L	Def. 4.5	4.1

$$u_i^h = \frac{C_i^h}{T_i}. \tag{4.2}$$

We will use u_i^h to decide whether a system is schedulable once tasks have been divided into τ^p and τ^h and $u_{i(j)}$ to determine how tasks should be divided.

Definition 4.2. The total utilizations of τ^p and τ^h are given by

$$U^p = \sum_{i=1}^{n^p} u_i^p \text{ and}$$
$$U^h = \sum_{i=1}^{n^h} u_i^h$$

respectively. To measure the total demand placed on the platform, we define *effective utilization* as

$$U^E = U^p + \frac{U^h}{2}.$$

U^h is halved to reflect each threaded task requiring only half a core at a time to execute. ◀

Example 4.1 shows why τ^p and τ^h must be scheduled separately.

Example 4.1. Suppose we attempt to schedule a task system τ using G-EDF. Let τ_1 be a threaded task and τ_2 a physical task such that at time t, a job of $\tau_1$with a deadline of $t+1$ is contending for a single core with a job of τ_2 with a deadline of $t+2$. We have three choices. First, giving τ_1 priority would respect G-EDF rules. However, if no other threaded task has an active job at time t, then doing so will cause the second threaded processor of the chosen core to be unused, negating any advantage of SMT. Second, giving τ_2 priority avoids wasting execution resources, but breaks G-EDF priority rules, potentially causing τ_1 to finish late. Third, co-scheduling the two jobs despite τ_2 being a physical task will cause τ_2 to have a longer than anticipated execution time, potentially causing it to finish late. None of these options are particularly satisfactory. ◊

The solution is to never allow this situation to happen: threaded and physical tasks must never contend for the same core. To enforce this rule and prevent a situation such as Example 4.1 from occurring, we divide platform π into *sub-platforms* π^p and π^h.

Definition 4.3. π^p is the sub-platform of π that schedules only tasks in τ^p. It includes $m^p = \lfloor U^p \rfloor$ fully available cores and one partially available core. Given a length-W interval, denoted a *window*, the partially available core belongs to π^p for $b^p W$ time units per window, where $b^p = U^p - \lfloor U^p \rfloor$. π^p can exist only if $U^p \leq m$. ◀

Definition 4.4. π^h is the sub-platform of π that schedules only tasks in τ^h. It has $m^h = m - \lceil U^p \rceil$ fully available cores and one core available for $b^h W$ time units per window, where $b^h = \lceil U^p \rceil - U^p$. Consequently, $m^h + b^h = m - U^p$. If $b^p > 0$, then $b^h = 1 - b^p$. ◀

We refer to the core shared by both platforms as the *shared core.* If there is no shared core, then $b^p = b^h = 0$. Otherwise, $b^p + b^h = 1$. Note that $m^p + b^p + m^h + b^h = m$ must hold.

Example 4.2. In Fig. 4.1, π^p is shown in dark gray and π^h in light gray. The figure depicts two separate platforms on the left and right; only the right platform has a shared core. The sub-platforms on the left

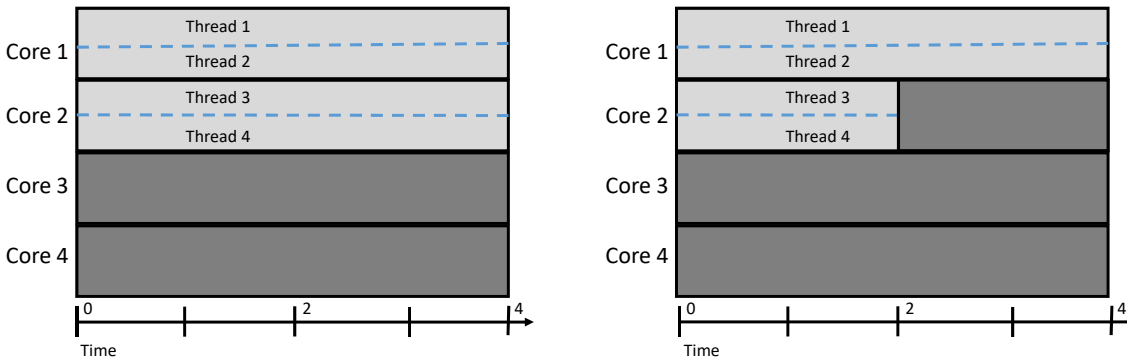

Figure 4.1: Two configurations of a platform supporting both SMT and non-SMT workloads.

are defined by by $m^p = 2$, $m^h = 2$, $W = 4$, and $b^p = b^h = 0$. The sub-platforms on the right are defined by $m^p = 2$, $m^h = 1$, $W = 4$, and $b^p = b^h = \frac{1}{2}$. $\Diamond$

When the only requirement is to establish a schedule for which tardiness is bounded, the value of W is theoretically irrelevant. When tardiness is taken into consideration, lower values will produce smaller tardiness bounds. However, lower values for W will also cause more frequent task preemptions and greater scheduling overheads, potentially leading to greater tardiness bounds in practice. In future work, we will examine the effects of different values for W.

We now give schedulability results for τ^p and τ^h individually and then combine those conditions to get an overall schedulability result. For the most part, we will focus on the case where a shared core exists. Our results are based on Devi and Anderson's EDF-high-low (EDF-HL) algorithm (Devi and Anderson, 2006). EDF-HL gives schedulability conditions and tardiness bounds for "low" SRT tasks that are scheduled according to G-EDF but are subject to interruption from periodic "high" hard real-time tasks, with at most one such task fixed on each processor. For our purposes, we can view τ^p as a set of low tasks scheduled on $m^p + \lceil b^p \rceil$ processors and subject to preemption by a single high task with period W and cost $b^h W$. This reflects the fact that, from the perspective of τ^p, work on the shared core is periodically preempted. Likewise, we can view τ^h as a set of low tasks scheduled on $2(m^h + \lceil b^p \rceil)$ processors that are periodically preempted by two high tasks, both with period W and cost $b^p W$. The following definitions apply to the EDF-HL results.

Definition 4.5. Devi and Anderson define τ_H as the set of all high tasks, τ_L as the set of all low tasks, $u_{max}(\tau_L)$ as the highest-utilization task within τ_L, U_{sum} as the total utilization of both τ_H and τ_L, U_H as the sum of all the utilizations of all tasks in τ_H, and U_L as the sum of the $min(\lceil U_{sum} \rceil - 2, n)$ largest utilization of tasks in τ_L.

We state an abridged version of Theorem 1 in (Devi and Anderson, 2006) here.

Theorem 4.1. *EDF-HL ensures a tardiness bound of at most B to every task τ_i of τ_L if $|\tau_H| \leq m$ and $U_{sum} \leq m$ and at least one of (4.3) or (4.4) holds.*

$$m - |\tau_H| - U_L > 0 \tag{4.3}$$

$$m - \max(|\tau_H| - 1, 0) u_{max}(\tau_L) - U_L - U_H > 0 \tag{4.4}$$

Returning to our problem of scheduling tasks with and without SMT seperately, our schedulabilty conditions rely on the following assumptions. These assumptions allow us to schedule τ^p and τ^h as if they both consisted of standard sporadic tasks.

Assumptions 4.1 will be addressed in Sec. 4.2. Assumption 4.2 can be fulfilled by following the timing procedures given in Sec. 3.1.

Assumption 4.1. Tasks have been divided into threaded and physical tasks such that $\forall \tau_i^p \in \tau^p, u_i^p \leq 1$ and $\forall \tau_i^h \in \tau^h, u_i^h \leq 1$ both hold. Without loss of generality, we assume that the tasks in each of the sets τ^p and τ^h are indexed in decreasing-utilization order, e.g., u_1^p (resp., u_1^h) is the largest utilization in τ^p (resp., τ^h).

Assumption 4.2. Costs for physical and threaded tasks have been determined.

Assumption 4.3. Physical tasks are not permitted to execute on processors in π^h.[1] Likewise, threaded tasks are not permitted to execute on processors in π^p.

With these assumptions in place, we can address the schedulability of τ^p and τ^h on the two platforms.

Lemma 4.1. *τ^p is schedulable on π^p under G-EDF such that all tasks have guaranteed bounded tardiness if (4.5) holds.*

$$U^p \leq m^p + b^p. \tag{4.5}$$

Proof. If $b^p = 0$, then the result restates the SRT feasibility condition for m identical, fully available processors that was given in Sec. 2.3.3. Because G-EDF is known to be SRT-optimal (Devi and Anderson, 2005), the result follows. If $m^p = 0$, then it can easily be shown that the system is schedulable only if $U^p \leq b^p$.

In the rest of the proof, we consider the remaining possibility, i.e., that $b^p > 0$ and $m^p > 0$ both hold. For this case, we show that Theorem 4.1 can be applied.

[1] When the shared core belongs to π^p, it supports a physical processor, not a threaded processor.

From the perspective of τ^p, there exists a set of low tasks τ^p with total utilization U^p, one high task with utilization b^h, and $m^p + 1$ processors. Thus, we want to apply Theorem 4.1 with the substitutions $m \leftarrow m^p + 1$, $\tau_L \leftarrow \tau^p$, $U_{sum} \leftarrow U^p + b^h$, and $|\tau_H| = 1$. With these substitutions, (4.5), and Definition 4.4, it is straightforward to see that both $|\tau_H| \leq m$ and $U_{sum} \leq m$ hold, as required by Theorem 4.1. We now show that (4.3) holds, from which bounded tardiness for the tasks in τ_L, i.e., those in τ^p, follows. To see this, note that from Definition 4.4 and $U_{sum} = U^p + b^h$, we have

$$
\begin{aligned}
U_L &= \sum_{i=1}^{min(\lceil U^p + b^h \rceil - 2, n^p)} u_i^p \\
&= \{\text{by Defs. 4.3 and 4.4, } U^p + b^h = m^p + 1\} \\
U_L &= \sum_{i=1}^{min(m^p - 1, n^p)} u_i^p \\
&\Rightarrow \{\text{because } u_i^p \leq 1 \text{ holds, by Assumption 4.1}\} \\
U_L &< m^p.
\end{aligned}
$$

From this inequality, we have $m - |\tau_H| - U_L = m^p + 1 - 1 - U_L > 0$, as required by (4.3). □

The schedulability condition for τ^h is complicated by the potential for two partially available processors.

Lemma 4.2. *τ^h is schedulable on π^h under GEDF such that all tasks have guaranteed bounded tardiness if (4.6) and at least one of (4.7) or (4.8) hold, where $u_{max}(\tau^h)$ denotes the maximum task utilization in τ^h.*

$$U^h \leq 2(m^h + b^h) \tag{4.6}$$

$$2m^h > \sum_{i=1}^{\min(2m^h, n^h)} u_i^h \tag{4.7}$$

$$2(m^h + b^h) - u_{max}(\tau^h) > \sum_{i=1}^{\min(2m^h, n^h)} u_i^h \tag{4.8}$$

Proof. As in the prior proof, the proof is straightforward if either $b^h = 0$ holds or $m^h = 0$ holds, so we focus on the remaining possibility, i.e, $m^h > 0$ and $a^h > 0$ both hold; note that the latter implies that $b^p > 0$ holds as well. As before, we will use Theorem 4.1. In this case, we are attempting to schedule a set of low tasks τ^h with total utilization U^h on $2(m^h + 1)$ processors given two high tasks, each with utilization b^p. Thus, we want to apply Theorem 4.1 with the substitutions $m \leftarrow 2(m^h + 1)$, $\tau_L \leftarrow \tau^h$, $U_{sum} \leftarrow U^h + 2b^p$, and $|\tau_H| = 2$.

With these substitutions, (4.6), and Definition 4.4, it is straightforward to see that both $|\tau_H| \leq m$ and $U_{sum} \leq m$ hold, as required by Theorem 4.1. In the rest of the proof, we show that, with these substitutions, (4.7) implies (4.3) and (4.8) implies (4.4), from which bounded tardiness for the tasks in τ_L, i.e., those in τ^h, follows.

To see that (4.7) implies (4.3), first note that, because m^h is an integer, we have $\lceil U_{sum} \rceil - 2 \leq \lceil m \rceil - 2 = \lceil 2(m^h+1) \rceil - 2 = \lceil 2m^h \rceil = 2m^h$. Therefore,

$$
\begin{aligned}
& 2m^h > \sum_{i=1}^{\min(2m^h, n^h)} u_i^h \\
\Rightarrow\ & \{\text{because } \lceil U_{sum} \rceil - 2 \leq 2m^h\} \\
& 2m^h > \sum_{i=1}^{\min(\lceil U_{sum} \rceil - 2, n^h)} u_i^h \\
=\ & \{\text{by the definition of } U_L \text{ in Def. 4.5}\} \\
& 2m^h > U_L,
\end{aligned}
$$

i.e., $2m^h - U_L > 0$ holds, which is equivalent to (4.3), since $m = 2(m^h+1)$ and $|\tau_H| = 2$.

To see that (4.8) implies (4.4), observe that

$$
\begin{aligned}
& 2(m^h + a^h) - u_{max}(\tau^h) > \sum_{i=1}^{\min(2m^h, n^h)} u_i^h \\
\Rightarrow\ & \{\text{reasoning as above}\} \\
& 2(m^h + b^h) - u_{max}(\tau^h) > U_L \\
=\ & \{\text{because } b^h = 1 - b^p\} \\
& 2(m^h + 1 - b^p) - u_{max}(\tau^h) > U_L, \\
=\ & \{\text{in our context } u_{max}(\tau^h) = u_{max}(\tau_L),\ |\tau_H| - 1 = 2,\ U_H = 2b^p, \text{ and } m = 2(m^h+1)\} \\
& m - \max(|\tau_H| - 1, 0) u_{max}(\tau_L) - U_H > U_L,
\end{aligned}
$$

which is equivalent to (4.4). □

A special case applies when there is no shared core.

Lemma 4.3. *If $b^h = 0$, then τ^h is schedulable on π^h under GEDF if and only if $U^h \leq 2m^h$ holds.*

Proof. With no shared core, the platform consists of $2m^h$ identical cores. The standard SRT feasibility test from (Devi and Anderson, 2005)—a system is feasible if and only if $U \leq m$ holds—applies. □

Our next step is to give a schedulability condition for τ^p and τ^h combined on π. This condition is a straightforward extension of the preceding lemmas, but it has the benefit of letting us focus on τ rather than on how π is partitioned.

Theorem 4.2. *Platform π can be partitioned such that τ^p is schedulable on π^p and τ^h is schedulable on π^h, both under G-EDF, if (4.9) and at least one of (4.10) or (4.11) hold.*

$$U^E \leq m \tag{4.9}$$

$$2(m - \lceil U^p \rceil) > \sum_{i=1}^{\min(2(m-\lceil U^p \rceil), n^p)} u_i^h \tag{4.10}$$

$$2(m - U^p) - u_1^h > \sum_{i=1}^{\min(2(m-\lceil U^p \rceil), n^p)} u_i^h \tag{4.11}$$

Proof. In order to define m^p and b^p so that $m^p + b^p = U^p$ holds, as in Definition 4.3, we merely require $U^p \leq m$ to hold, and by Definition 4.2, this is implied by (4.9). Note that $m^p + b^p = U^p$ satisfies Condition (4.5) in Lemma 4.1.

Schedulability of τ^p on π^p is implied by (4.9):

$$\begin{aligned}
U^E &\leq m \\
&= \{\text{by Def. 4.2, } U^E = U^p + \frac{U^h}{2}\} \\
U^p &\leq m \\
&= \{\text{by Def. 4.3, } m^p + b^p = U^p\} \\
U^p &= m^p + b^p,
\end{aligned}$$

which is the condition for τ^p per Lemma 4.1.

We next show that (4.9) implies Condition (4.6) of Lemma 4.2. To see this, observe that, by Definition 4.2, $U^E \leq m \Rightarrow \frac{U^h}{2} \leq m - U^p$. Also, by Definition 4.4, $m^h + b^h = m - U^p$. Putting these facts together, we have $U^h \leq 2(m^h + b^h)$, which is (4.6).

We conclude the proof by showing that (4.10) is equivalent to Condition (4.7) of Lemma 4.2, and that (4.10) is equivalent to Condition (4.8) of Lemma 4.2. To see the former, note the following.

$$\begin{aligned} 2(m - \lceil U^p \rceil) &> \sum_{i=1}^{\min(2(m-\lceil U^p \rceil), n^h)} u_i^h \\ &= \{\text{by Def. 4.4, } m - \lceil U^p \rceil = m^h\} \\ 2m^h &> \sum_{i=1}^{\min(2m^h, n^h)} u_i^h \end{aligned}$$

Similarly, to see that (4.11) holds, note the following.

$$\begin{aligned} 2(m - U^p) - u_1^h &> \sum_{i=1}^{\min(2(m-\lceil U^p \rceil), n^h)} u_i^h \\ &= \{\text{by Def. 4.4, } m - \lceil U^p \rceil = m^h.\} \\ 2(m^h + b^h) - u_1^h &> \sum_{i=1}^{\min(2m^h, n^h)} u_i^h. \end{aligned}$$

Having verified all conditions of Lemmas 4.1 and 4.2, we conclude that τ^p is schedulable on π^p and τ^h is schedulable on π^h. □

Again, a special case applies if U^p is integral.

Corollary 4.1. *If U^p is integral, then both τ^p and τ^h are schedulable on their respective sub-platforms under GEDF so long as $U^E \leq m$ holds.*

Proof. Similar to the proof of Lemma 4.3. □

It is not strictly necessary that π^p be defined as we do here. If we allow other design considerations, such as maximizing cache affinity or minimizing tardiness, different platform definitions may be preferable, but we defer those possibilities to future work.

By themselves, the results of this section are not very useful, since there are an exponential number of possible ways to partition π. In the next section, we show how to efficiently find τ^p and τ^h that will be schedulable under Theorem 4.2.

4.2 Dividing the Tasks

In this section we give algorithms for determining which tasks should and should not use SMT. In a system of n tasks, there are 2^n possible ways to partition the system into τ^p and τ^h. We attempt to quickly find a good partition by heuristically minimizing U^E. In many cases, doing so will minimize the number of cores needed to schedule a given system.

Oblivious scheduling. We first work through a simple example of dividing a task system and then formalize that approach into what we term *symbiosis-oblivious partitioning*.[2] We then show how our approach can be improved. Symbiosis-oblivious partitioning is based a simple rule:

Proposition 4.1. τ_i should be assigned to τ^h if and only if

$$\max_{\forall j \neq i} u_{i(j)} \leq 1 \text{ and}$$
$$\max_{\forall j \neq i} u_{i(j)} \leq \frac{u_i^p}{2} \text{ hold.}$$

Furthermore, if $n^h = 1$ holds, then all tasks must be scheduled without SMT. ◀

If we allowed for exactly one threaded task, then the "threaded" task would in fact always execute without SMT, giving no overall schedulability benefits. Algorithm 5 implements Prop. 4.1.

Example 4.3. Table 4.3 gives utilizations for a system of five tasks. Without SMT, the system has total utilization of

$$\sum_{i=1}^{5} u_i^p = 3.1$$

and requires four cores to schedule. When we apply Prop. 4.1, τ_3, τ_4, and τ_5 are assigned to τ^p; τ_1 and τ_2 are assigned to τ^h. The resulting system has effective utilization of

$$\begin{aligned} & U^p + \frac{U^h}{2} \\ & = 0.6 + 1.0 + 0.3 + \frac{1.0 + 0.9}{2} \\ & = 2.85 \end{aligned}$$

[2]The terms symbiosis-oblivious and symbiosis-aware scheduling were previously used by Jain et al.(Jain et al., 2002).

Table 4.3: Sample task system with five tasks.

measured task	u_i^p	$u_{i(j)}$ τ_1	τ_2	τ_3	τ_4	τ_5	$\max_{\forall j \neq i} u_{i(j)}$
τ_1	0.6	NA	0.8	0.9	1.0	1.0	1.0
τ_2	0.6	0.8	NA	0.9	0.9	0.9	0.9
τ_3	0.6	0.7	0.7	NA	1.1	0.9	1.1
τ_4	1.0	1.2	1.4	1.2	NA	1.5	1.5
τ_5	0.3	1.0	0.9	0.7	0.7	NA	0.7

and can be scheduled on three cores. ◊

Symbiosis-oblivious partitioning can be effective at reducing the number of cores needed to schedule a system. However, it overlooks a way to achieve further reductions.

Example 4.4. In Example 4.3, we decided that τ_3 should be assigned to τ^p. However, $u_{3(j)} > 1$ holds only for $j = 4$. With τ_4 in τ^p, we can reassign τ_3 to τ^h. This partition, with $\tau^h = \{\tau_1, \tau_2, \tau_3\}$ and $\tau^p = \{\tau_4, \tau_5\}$ has effective utilization of

$$\begin{aligned} & U^p + \frac{U^h}{2} \\ & = 1.0 + 0.3 + \frac{0.9 + 0.9 + 0.7}{2} \\ & = 2.55. \end{aligned}$$

Note that we use $u_1^h = 0.9$ and $u_2^h = 0.7$; because τ_1 and τ_2 will not experience interference from τ_4 or τ_5, we need not consider their effects. ◊

Algorithm 6 formalizes the scheduling approach of Example 4.4. The algorithm seeks to minimize U^E by repeatedly moving a task from τ^p to τ^h, or vice versa, to give the greatest decrease in U^E. It does so until either a specified maximum number of attempts has been made or it reaches a partition that cannot be improved by the movement of any single task. The algorithm is not optimal, even given an unlimited number of attempts, as there may exist partitions of τ that cannot be improved by moving any one task but can be improved by moving two or more tasks.

Algorithm 5 Oblivious Partitioning

for all $\tau_i \in \tau$ **do**
  $C_i^h \leftarrow \max_{\forall j \neq i} C_{i(j)}$
  if $C_i^h \leq T_i$ **and** $\frac{C_i}{C_i^h} \geq 2$ **then**
    $\tau^h \leftarrow \tau^h \cup \tau_i$
  else
    $C_i^p \leftarrow C_i$
    $\tau^p \leftarrow \tau^p \cup \tau_i$
  end if
end for
if $|\tau^h| < 2$ **then**
  $\tau^p \leftarrow \tau^p \cup \tau^h$
  $\tau^h \leftarrow \emptyset$
end if
return τ^p, τ^h

Algorithm 6 will never create a partition that cannot be scheduled even given an unlimited number of cores—this situation would occur if τ^h is such that $u_i^h > 1$ holds for at least on τ_i—or that will cause τ^h to have exactly one component tasks. We refer to a task system with these two properties as being *legal.*

Definition 4.6. A partition of τ into τ^p and τ^h is *legal* if and only if $\forall\ \tau_i^h \in \tau^h$, $u_i^h \leq 1$ and $|\tau^h| \neq 1$ hold. ◄

Algorithm 6 assumes, and maintains as an invariant, that the partition is legal. To begin Algorithm 6, τ must already be in a legal partition. We propose three ways to achieve this. First, in the *greedy-threaded* approach, we begin with all tasks in τ^h and then place into τ^p all tasks for which any possible C_i^h value will give $u_i^h > 1$. Intuitively, putting tasks in τ^h whenever possible should be beneficial, so we should start with as many tasks in τ^h as possible.

Second, in the *greedy-physical* approach, we start with all tasks in τ^p apart from the single pair of tasks that will give the greatest decrease to U^E. This can be done by defining the decrease to U^E associated with a single pair of tasks (τ_i, τ_j) as

$$\forall\ (i,j), \Delta(i,j) = u_i^p + u_j^p - \frac{1}{2}(u_{i(j)} + u_{j(i)})$$

and adding to τ^h the pair of tasks that maximize $\Delta(i,j)$ subject to $u_{i(j)} \leq 1$ and $u_{j(i)} \leq 1$.

When τ_i and τ_j are placed into τ^h, u_i^p and u_j^p are no longer part of U^p and can be subtracted from U^E. However, we must add half of the new U^h value, $u_{i(j)} + u_{j(i)}$ to U^E. We expect this approach will be more efficient than the first one in task systems where u_i^p is typically large or $u_{i(j)}$ is typically small, since there

will be relatively few tasks that can be placed in τ^h, making it more efficient to begin with the majority of tasks in τ^p. If no satisfactory pair of tasks exists, then we conclude that SMT should not be used.

Algorithm 6 Greedy Partitioning

Require: τ partitioned such that $\forall \tau_i \in \tau^h, u_i^h \leq 1$ and $|\tau^h| \geq 2$

for $\ell \leftarrow 1 ... maxLoops$ **do**
  ▷ Identify best move from τ^p to τ^h
  for all $\tau_i \in \tau^p$ **do**
    $C_i^h = \max_{\forall \tau_j \in \tau^h, j \neq i} C_{i(j)}$
    $u_i^h = \frac{C_i^h}{T_i}$
    if $u_i^h > 1$ **then**
      continue
    end if
    ▷ Calculate how adding τ_i to τ^h will affect tasks already in τ_h
    if moving τ_i to τ^h will cause $u_j^h \geq 1$ for any $\tau_j \in \tau^h$ **then**
      continue
    end if
    $I(\tau_i^h) \leftarrow$ total increase in util. of tasks already in τ^h caused by moving τ_i
    ▷ $\Delta(i)$ gives decrease to U^E caused by moving τ_i.
    $\Delta(i) \leftarrow u_i^p - \frac{u_i^h + I(\tau_i^h)}{2}$
  end for
  ▷ Identify best move from τ^h to τ^p
  if $|\tau^h| > 2$ **then**
    for all $\tau_j \in \tau_h$ **do**
      $D(\tau_j^h) \leftarrow$ total decrease in util. of tasks already in τ^h caused by moving τ_j
      ▷ $\Delta(j)$ gives decrease to U^E caused by moving τ_j.
      $\Delta(j) \leftarrow \frac{u_j^h + D(\tau_j^h)}{2} - u_j^p$
    end for
  end if
  if no task has a positive Δ value **then**
    break
  end if
  Move task with maximum Δ to other subsystem and update threaded costs
end for
return(τ^p, τ^h)

Third, in the *greedy-mixed* approach, we first apply Prop. 4.1 via Algorithm 5 and then use the partition given by doing so as our starting point. Intuitively, Prop. 4.1 by itself should give a partition with a lower U^E value than either of the other two approaches, so using it is a starting point should yield better results. As with the greedy-physical approach, if Algorithm 5 places no tasks in τ^h, then we conclude that SMT should not be used. We compare these three approaches in our schedulability experiments, presented in Sec. 5. We

found that for all three versions of Algorithm 6, there existed task systems that were schedulable according to that version alone. However, the greedy-physical approach had the best overall performance.

Regardless of how we obtain the initial legal partition for Algorithm 6, the **for** loop of lines 3 through 16 determines, for every τ_i in τ^p, the benefit of moving that task to τ^h. Line 4 tests what C_i^h would be if τ_i were in τ^h. Lines 10 through 13 calculate the change to tasks already in τ^h caused by moving τ_i, and line 15 gives the total change to U^E caused by moving τ_i to τ^h.

The **for** loop of lines 19 through 23 determines the benefit of moving τ_j to τ^p, for every τ_j currently in τ^h. Line 20 gives the change to tasks remaining in τ^h caused by moving τ_j, and line 22 gives the total change to U^E caused by moving τ_j to τ^p. The **if** of line 25 guarantees that no task will be moved unless moving that task will decrease U^E, preventing the algorithm from placing τ into any one partition more than once.

4.3 Schedulability Studies

To evaluate our work, we conducted a large scale synthetic-task schedulability study. A synthetic task is defined by parameters specifying its period, cost, and, in our case, costs when co-scheduled with other tasks. In a synthetic-task schedulability study, scheduling algorithms are evaluated by comparing their ability to schedule systems of synthetic tasks. The advantage of this approach is that a nearly-unlimited number of task systems can be created with exactly the properties we are most interested in testing. For example, we can evaluate an algorithm's system to schedule harmful tasks, as discussed in Section 3.1.5, by creating task systems that use different values of h while holding all other values constant.

In our study, we evaluate the ability of the Algorithms of Section 4.2 to schedule task systems that would be unschedulable without SMT. We first use each of the four algorithms to divide each task system into τ^p and τ^h and then use Theorem 4.2 to decide if the resulting partitioned system is schedulable.

4.3.1 Schedulability Study Overview

Whether or not one synthetic system can be scheduled is not very informative. For this reason, we consider scheduling *scenarios* defined by a per-task utilization range (without SMT), and the number of cores m on the (synthetic) target platform, and a model for SMT interaction. We give details on these parameters in Section 4.3.2. For each scenario, we created synthetic systems with total utilizations ranging from m to $2m$. Each scenario is summarized in a graph that shows the percentage of systems schedulable at each utilization

level in increments of 0.1 for four and eight core platforms or 0.2 for 16-core platforms. Each point on our graphs corresponds at least 500 task systems. Each scenario is further summarized by its *relative schedulable area*, or RSA.

Definition 4.7. The *relative schedulable area* is defined as the area under the curve in a schedulability graph divided by the core count m for the scenario. ◄

In calculating RSAs, we assume that all systems with utilization no greater than m are schedulable. This assumption is based on the ability of G-EDF to correctly schedule all SRT systems for which $U \leq m$ holds; we did not actually test systems with $U < m$. A G-EDF scheduler not using SMT will have an RSA of 1.0; RSAs therefore indicate how much additional work can be accommodated on a given platform by enabling SMT. A scheduler capable of scheduling all systems with $U \leq 1.2m$ and no systems with $U > 1.2m$ would have an RSA of 1.2.

4.3.2 Schedulability Test Parameters

We use five parameters to define each scenario. The first parameter, core count (m), specifies the number of cores in our target platform. Possible values are four, eight, and sixteen.

The second parameter determines task utilizations before accounting for SMT. In scenarios using the *light* distribution, task utilizations are assigned from the Uniform probability distribution $(0, 0.4)$. In the *medium*, *wide*, and *heavy* distributions, utilizations are assigned from the Uniform distributions (0.3, 0.7), (0, 1), and (0.6, 1), respectively.

The remaining three parameters govern the effects of SMT on execution times. Recall from Definition 3.3 a task's multithreading score is given by

$$M_{i(j)} = \frac{C_{i(j)} - C_i}{C_i},$$

i.e. $C_{i(j)} = C_i \cdot (M_{i(j)} + 1)$.

We allow three possible values for the overall expected value of all $M_{i(j)}$ values, denoted μ: 0.2 (optimistic), 0.4 (mid-range), and 0.6 (pessimistic). The parameter h (Definition 3.6) gives the probability that a specific tasks will be more harmful, i.e., cause the execution times of other tasks to increase more. Possible values for h are 0, 0.125, and 0.25. Finally, the distribution parameter determines whether $C_{i(j)}$ is deterministic given τ_i and τ_j (*fixed*) or involves an additional randomization step (*exponential*).

Table 4.4: Summary of parameters used to define schedulability scenarios

Parameter description	Parameter symbol	Possible Values
core count	m	4, 8, 16
per-task util. distribution	none	U(0, 0.4) (light); U(0.3, 0.7) (medium); U(0, 1) (wide); U(0.6, 1.0) (heavy)
expected $M_{i(j)}$	μ	0.2, (optimistic); 0.4, (mid-range); 0.6 (pessimistic)
probability of task being harmful	h	0, 0.125, 0.25
distribution for $M_{i(j)}$ given τ_i and τ_j	none	fixed exponential

Parameter choices are summarized in Table 4.4. In total, our parameter choices allow us to define 216 scheduling scenarios, with each scheduling scenario consisting of tens of thousands of task systems. Creating and testing all systems in our scenarios required approximately six weeks of CPU time.

4.3.3 Schedulability Test Results

Our full set of graphs and scenario results is included in Appendix A. Here, we use a subset of our graphs to illustrate key observations from our data. The first two lines of each graph's title indicate the scenario it represents. The last line of the title gives the RSA for that scenarios best-performing partitioning algorithm. These values ranged from a minimum of 1.03 to a maximum of 1.54. Mean and average RSAs were both 1.21.

Example 4.5. Figure 4.3 depicts scheduling systems of medium-weight tasks on a four core platform (first line of graph title) give $\mu = 0.6$, $h = 0$, and the fixed distribution is used for $M_{i(j)}$ (second line of title). The greedy-physical algorithm gave the best performance, with an RSA of 1.29 (third line of title). The greedy-threaded, greedy-mixed, and oblivious partitioning algorithms all were slightly less able to partition task systems into something that could be scheduled. The greedy physical algorithm was able to schedule almost all task systems with $U \leq 4.25$, roughly half of task systems with $U = 5.25$, and almost no systems with $U > 6$. $\Diamond$

Observation 4.1. *When using SMT, a large majority of task systems that would require $m+1$ cores without SMT, i.e., with utilization in the range $(m, m+1]$, could be scheduled on m cores.* Table 4.5 gives the

Table 4.5: Schedulability of systems with U in range $(m, m+1]$.

	best scenario	mean scenario	median scenario	worst scenario
$m = 4$	99.9%	66.2%	78.0%	9.6%
$m = 8$	100%	83.8%	86.1%	20.1%
$m = 16$	100%	87.7%	92.3%	26.3%

percentage of task systems with utilizations in this range that can be scheduled on m cores under various scenarios. The best scenario in the context of Table 4.5 is the scenario where the maximum number of systems with $m < U \leq m+1$ are schedulable. Likewise, the worst scenario is the scenario with the fewest schedulable systems in that utilization range.

Observation 4.2. *SMT is beneficial even if $M_{i(j)}$ values are greater than what we observed.* Figure 4.2 shows an average scenario among those with pessimistic μ values. The graph has an RSA of 1.15. In the best cases, scenarios with $\mu = 0.6$ can have RSA values as high as 1.29; 4.3 shows the best-case scenario for $\mu = 0.6$

Smaller values of μ naturally yield greater RSAs. Figs. 4.4 through 4.7 show average and best scenarios for $\mu = 0.4$ and $\mu = 0.2$.

Observation 4.3. *High utilization tasks have a negative impact on SMT.* In the worst case, using SMT with heavy per-task utilizations shows few benefits (Figure 4.8). Even an average scenario given heavy per-task utilization (Figure 4.9) is worse than the worst scenarios for both light and medium per-task utilization (Figs. 4.10 and 4.11).

The presence of high-utilization tasks appears more important than average per-task utilization. The medium and wide distributions both have average utilization of 0.5, but the medium average case (Figure 4.12) is noticeably better than the wide average case (Figure 4.13).

The importance of per-task utilization is not surprising; if a task has utilization close to 1.0 even without SMT, even a small difference between C_i^h and C_i can cause $u_i^h > 1.0$ told hold, making scheduling impossible if τ_i uses SMT.

Observation 4.4. *Scenarios using the fixed $M_{i(j)}$ distribution saw greater benefits from SMT than those with the exponential distribution.* Figs. 4.14 and 4.15 show average cases for both distributions, with all other parameters held constant. All of our best scenarios depicted (Figs. 4.3, 4.5, and 4.7) used the fixed distribution, whereas all of our worst scenarios depicted (Figs. 4.8, 4.10, 4.11) use the exponential distribution.

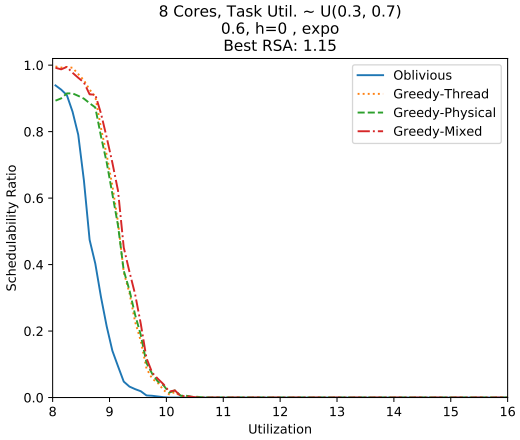

Figure 4.2: An average scenario for $\mu = 0.6$

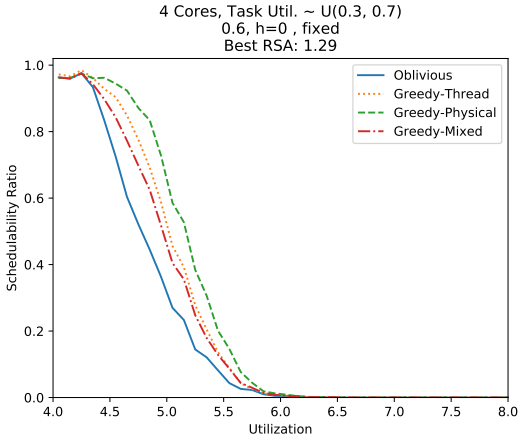

Figure 4.3: The best scenario for $\mu = 0.6$

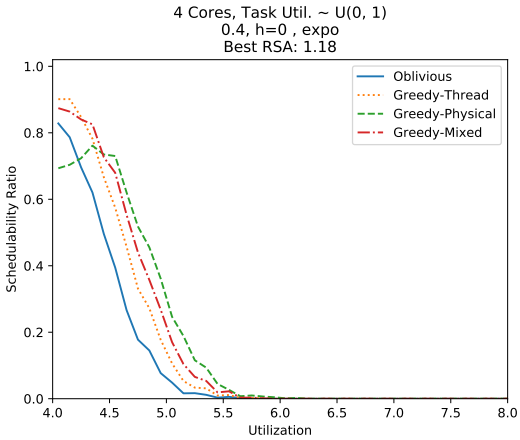

Figure 4.4: An average scenario for $\mu = 0.4$

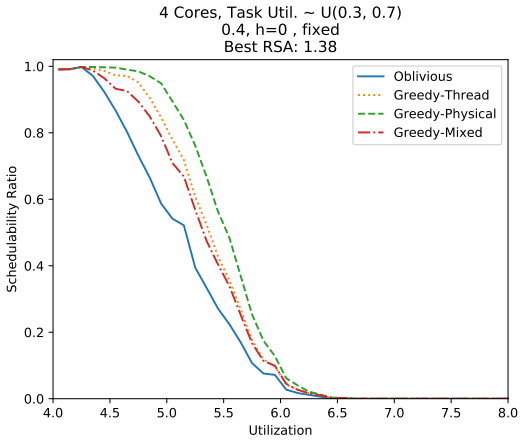

Figure 4.5: The best scenario for $\mu = 0.4$

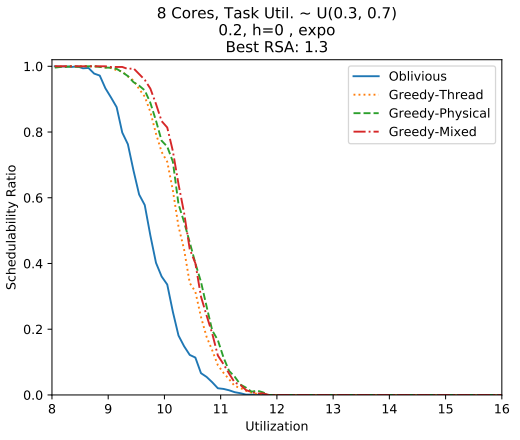

Figure 4.6: An average scenario for $\mu = 0.2$

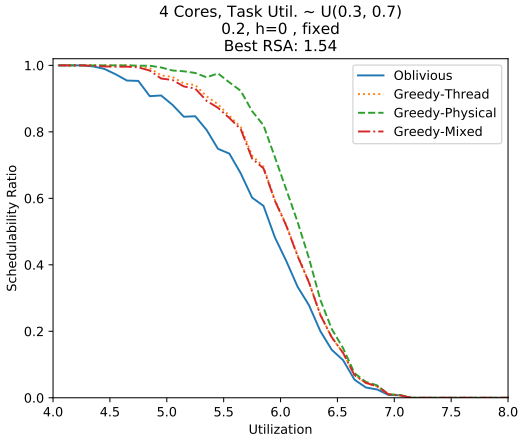

Figure 4.7: The best scenario for $\mu = 0.2$

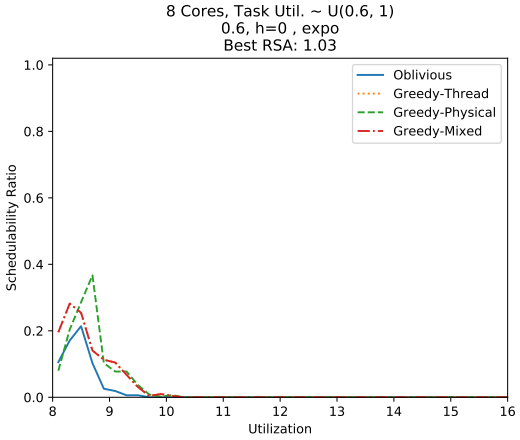

Figure 4.8: The worst scenario for heavy per-task utilization.

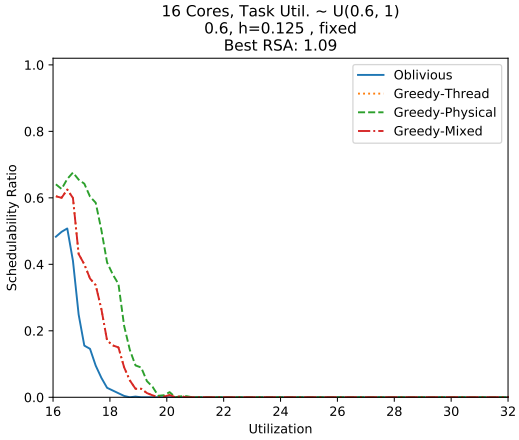

Figure 4.9: An average scenario for heavy per-task utilization.

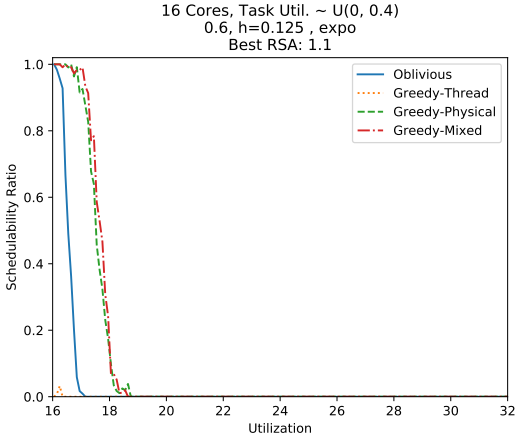

Figure 4.10: The worst scenario for low per-task utilization.

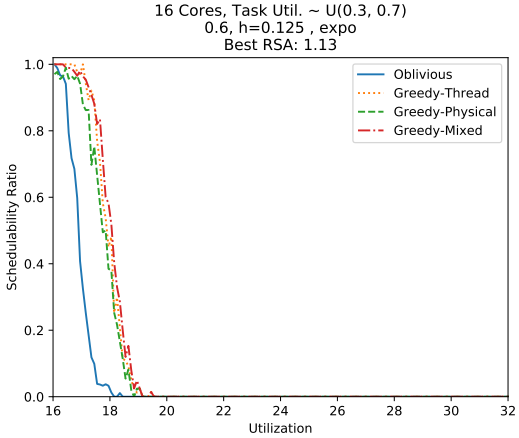

Figure 4.11: The worst scenario for medium per-task utilization..

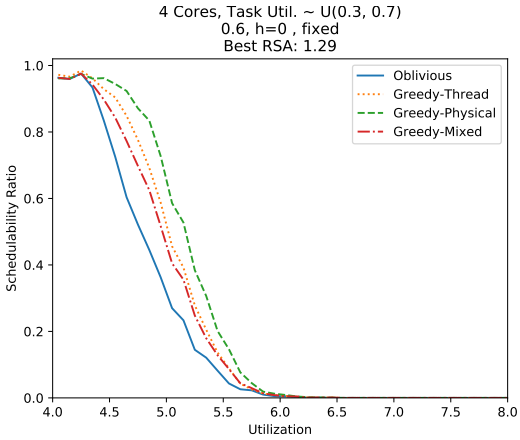

Figure 4.12: An average scenario for medium per-task utilization.

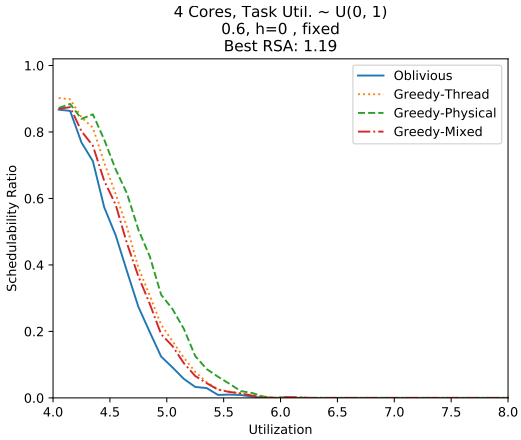

Figure 4.13: An average scenario for wide per-task utilization.

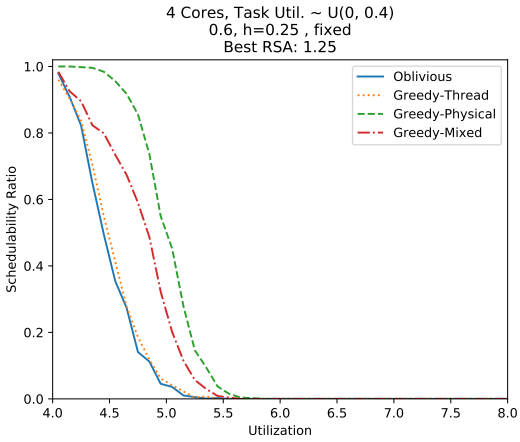

Figure 4.14: An average scenario for the fixed per-task $M_{i(j)}$ distribution.

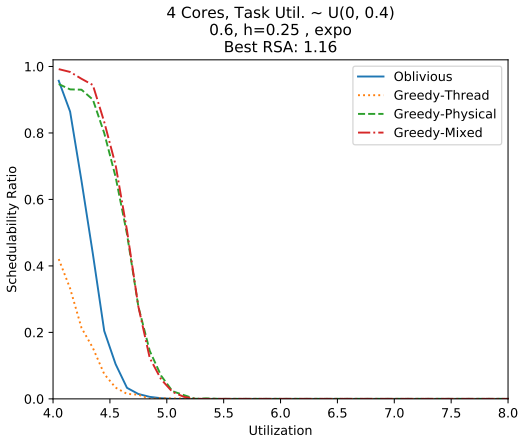

Figure 4.15: An average scenario for the exponential per-task $M_{i(j)}$ distribution.

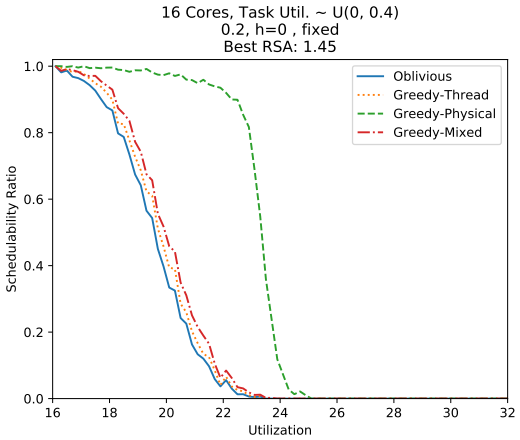

Figure 4.16: The best 16-core scenario; $h = 0$.

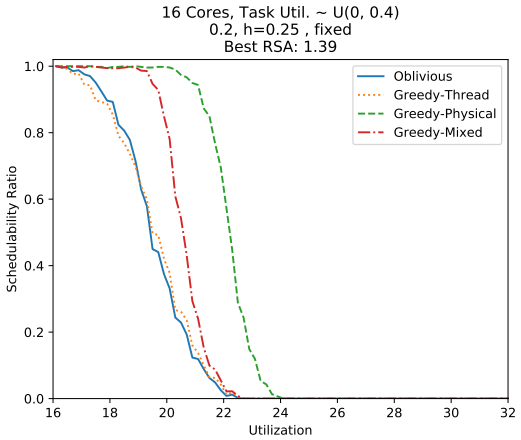

Figure 4.17: Scenario with $h = 0.25$.

More broadly, the fixed and exponential distributions can be interpreted as stand-ins for how much $M_{i(j)}$ varies as j changes. When $M_{i(j)}$ is near-constant with respect to j, then using SMT is more likely to be either always helpful or always harmful.

Observation 4.5. *Increasing h appears to have a slight harmful effect.* Scenarios with $h = 0$, $h = 0.125$, and $h = 0.25$ have average RSAs of 1.22, 1.21, and 1.21, respectively, and maximum RSAs of 1.53, 1.51, and 1.49. The effects of h become more pronounced as core count increases, presumably because each task can then cause interference to more other tasks.

Figure 4.16 shows the best 16-core scenario; $h = 0$. Figure 4.17 shows a slightly inferior scenario with $h = 0.25$ and otherwise identical parameters.

Observation 4.6. *Increasing core count has a slight negative effect.* Scenarios with four, eight, and 16 have mean RSAs of 1.23, 1.21, and 1.19, respectively, and maximums of 1.53, 1.49, and 1.45. Figure 4.7 is both the best overall scenario and the best 4-core scenario; Figure 4.16 is the best 16-core scenario. As already mentioned, more cores imply more tasks that may have the opportunity to interfer with one another.

Observation 4.7. *No single algorithm was consistently dominant.* Of the three forms of the greedy algorithm, the greedy-physical algorithm gave the best overall performance; it was dominant in 138 scenarios, or nearly two-thirds of the time. The greedy-mixed algorithm was dominant in 60 scenarios, all of which used the exponential rather than fixed $M_{i(j)}$ distribution. Note that the greedy-mixed algorithm always dominates the oblivious algorithm. The greedy-threaded algorithm was dominant in 18 out of 216 scenarios, all of which used the heavy per-task utilization range. However, in all of these 18 scenarios, the greedy-threaded algorithm was either identical to the greedy-mixed algorithm or only gave a very slight improvement.

4.4 Conclusions and future work

In this chapter, we have given a schedulability test and four related algorithms for using SMT to support SRT systems. We have shown, within the context of our schedulabilty study, that SMT can, in the best cases, allow a given platform to schedule systems with utilization 1.5 times greater than what it could schedule without SMT.

In future work, we plan to develop partitioning algorithms that account for cache affinity and use SMT to decrease tardiness rather than simply guaranteeing bounded tardiness.

CHAPTER 5: SCHEDULING HARD REAL-TIME SYSTEMS WITH SMT

In this chapter, we give two approaches to scheduling periodic HRT systems. First, in Section 5.1, we describe the CERT-MT (Controlled Execution of Real-Time with Multi-Threading) scheduler. In Section 5.2, we give the results of testing CERT-MT with a synthetic-task schedulability study. We show that CERT-MT can be effective at leveraging SMT in small platforms, but that scalability problems make it less useful for large platforms. Second, in Section 5.3, we introduce an alternative scheduler that is more scalable and arguably easier to implement. We evaluate this scheduler in Section 5.4.

Both schedulers of this chapter make use of mathematical optimization programs. To solve them, we used Gurobi Optimizer (Gurobi Optimization LLC, 2023), a commercial optimization program solver with free academic licensing.

The notation used in this chapter is summarized in Table 5.1. The work of this chapter is a continuation of that in (Osborne and Anderson, 2020; Osborne et al., 2020).

5.1 CERT-MT: Controlled Execution of Real-Time with Multithreading

In this section, we present the CERT-MT scheduler, a cyclic executive (CE) scheduler (see Section 2.3.2) that allows for SMT. CERT-MT allows for RSAs as high as 1.34, but suffers from significant scalability problems.

We repeat the definition of simultaneous co-scheduling (Definition 3.7) for easy reference.

Definition 5.1. We say that two jobs are *simultaneously co-scheduled* if both begin execution simultaneously on hardware threads belonging to the same physical core, and when one job completes, the remaining job continues on the same core until complete. No other job can be scheduled on the core until both jobs have completed. We use $\tau_{i.a:j.b}$ to denote the simultaneously co-scheduled jobs $\tau_{i.a}$ and $\tau_{j.b}$. ◀

Table 5.1: Summary of notation used in Chapter 5.

Symbol	Description	Reference	Sections Used
$\tau_{i.a:j.b}$	Simultaneously co-scheduled HRT jobs	Def. 3.7	3.2, 5.1
$C^{(1)}_{i(j)}$	Cost of HRT task τ_i co-scheduled with τ_j	Def. 3.9	3.2, 3.3, 5.1
$\phi(\ell)$	Frame size on core ℓ	Def. 5.2	5.1
H	Hyperiod length	Def. 2.6	2.3, 5.1
$C^{+}_{i(j)}$	$\max(C^{(1)}_{i(j)}, C^{(1)}_{i(j)})$	Def. 5.3	5.1, 5.3
$r(i.a, j.b)$	$\max(T_i \cdot (a-1), T_j \cdot (b-1))$	Def. 5.4	5.1
$d(i.a, j.b)$	$\min(T_i \cdot a, T_j \cdot b)$	Def. 5.4	5.1
$x(i.a, j.b, \ell, g)$	Decision variable for $\tau_{i.a:j.b}$	Def. 5.6	5.1
$\tau_{i:j}$	Paired tasks	Def. 5.8	5.3
$x_{i:j}$	Decision variable for $\tau_{i:j}$	Def. 5.10	5.3
b_i	Blocking term for τ_i.	Def. 2.8	2.3, 5.3
$C^{-}_{i(j)}$	$\min(C^{(1)}_{i(j)}, C^{(1)}_{i(j)})$	Def. 5.11	5.3
$\min\left(\frac{C_i}{C_j}, 1\right)$	Adjusted Score Ratio	Def. 3.18	3.3, 5.2, 5.4
$f_i(\min\left(\frac{C_i}{C_j}, 1\right))$	Linear regression function	Def. 3.19	3.3, 5.2, 5.4
RSA	Relative Schedulable Area	Def. 4.7	4.3, 5.2, 5.4

5.1.1 Scheduling Definition and Limitations

To ensure that CERT-MT operates within the bounds of the timing analysis work in Section 3.2, we require that all jobs using SMT must be simultaneously co-scheduled and executed non-preemptively; scheduler interruptions are included as preemptions. As we did not consider the effects of preemption in Section 3.2, allowing preemptions while using SMT here would lack justification. We do allow preemptions when not using SMT. In addition, as stated in Section 3.3, we do not allow tasks with baseline execution times that differ by more than a factor of 10 to be co-scheduled.

Requiring non-preemptive scheduling to stay within the bounds of our timing analysis can make otherwise schedulable systems unschedulable, as shown in Example 5.1 below. The example does not explicitly reference SMT or co-scheduled jobs; Definitions 5.3 and 5.4, presented later in this subsection, will allow us to treat co-scheduled jobs as a single schedulable unit.

Example 5.1. Let τ be a task system with periods in the set $\{10, 20\}$ such that $C_i > 10$ holds for at least one task. Consider the frame size f[1] needed to execute τ non-preemptively. If $f > 10$ holds, then no job with $T_i = 10$ can be scheduled, since the first frame will not complete until after the deadline of the first job, so

[1]Unrelated to the linear regression function discussed in Section 3.3.

such a job could miss its deadline. If $f \leq 10$ holds, then no job with $C_i > 10$ can be scheduled, since the frame boundary causes a preemption by the scheduler. ◊

To facilitate the scheduling of non-preemptive jobs, we allow the frame size to be defined per-core, rather than requiring one frame size for the entire platform. We replace the notation f for a platform-wide frame size—the standard in CE scheduling—with $\phi(\ell)$.

Definition 5.2. A *frame* is a time interval of length $\phi(\ell)$ on core ℓ. Frames are indexed using g, starting from $g = 1$, such that the g^{th} frame on core ℓ starts at time $(g-1) \cdot \phi(\ell)$ and ends at time $g \cdot \phi(\ell)$. We require that no frame extends beyond the hyperperiod[2] boundary, allowing the schedule to repeat every hyperperiod. The last frame per hyperperiod on core ℓ has index $g = \left\lfloor \frac{H}{\phi(\ell)} \right\rfloor$. ◄

As a consequence of allowing multiple frame sizes, frames with the same indices may cover different lengths of time on different cores. For example, suppose that on a two-core system, $\phi(1) = 10$ and $\phi(2) = 20$. Frames 1, 2, and 3 of core 1 would start at times 0, 10, and 20, but frames 1, 2, and 3 of core 2 would start at times 0, 20, and 40. In addition, the maximum valid frame index varies based on $\phi(\ell)$; since we only need to define a schedule for the first hyperperiod, we do not consider frames for which $g \cdot \phi(\ell) > H$ holds.

We define three scheduling parameters for co-scheduled job pairs: *outer cost*, *joint release*, and *joint deadline*. These parameters allow us to treat pairs of jobs as schedulable entities.

Definition 5.3. The *outer cost* to simultaneously execute jobs of τ_i and τ_j is given by $C^+_{i(j)}$, defined as the execution time for both jobs assuming they begin simultaneously, i.e.,

$$C^+_{i(j)} = \max\left(C^{(1)}_{i(j)}, C^{(1)}_{j(i)}\right).$$

If $i = j$, then $C^+_{i(j)} = C_i$, indicating solo execution for τ_i. Jobs with nothing co-scheduled are *solo jobs*. ◄

We make the pessimistic assumption that when co-scheduled, *both* jobs require execution time $C^+_{i(j)}$ to complete; neither job gets credit for completing until both jobs have done so. This assumption simplifies our scheduling problem considerably.

Definition 5.4. Given $\tau_{i.a:j.b}$, the *joint release* and *joint deadline* are given, respectively, by

$$r(i.a, j.b) = \max(T_i \cdot (a-1), T_j \cdot (b-1)) \text{ and}$$

[2]Recall from Definition 2.6 that the hyperperiod is denoted by H

$$d(i.a, j.b) = \min(T_i \cdot a, T_j \cdot b).$$

For the case where $i = j$ and $a = b$—i.e., a solo job—the r and d terms are simply the job's release time and absolute deadline. ◄

In order for both jobs of a pair to finish on time, the pair must begin no sooner than $r(i.a, j.b)$—both jobs must have been released—and must finish no later than $d(i.a, j.b)$—neither job can miss its deadline.

5.1.2 Creating a Schedule

In this subsection, we review what it means for a CE scheduler to be correct and generalize standard CE rules to allow for the use of SMT and varied frame sizes. A CE schedule is correct if the rules previously stated in Definition 2.7, adapted from Baker and Shaw (Baker and Shaw, 1989), hold. We repeat the definition here.

Definition 5.5. A CE schedule is *correct* if over the course of each hyperperiod: (i) all jobs are scheduled; (ii) any non-preemptable job is scheduled in one frame; (iii) every job completes in a frame that ends no later than its deadline; (iv) no job executes in a frame that begins before its release; (v) the total execution time scheduled in each frame is no greater than the frame size; and (vi) no job executes in parallel with itself. ◄

If we allow a particular job to execute on multiple cores, rule (vi) of Definition 2.7 becomes quite challenging. For this reason, we require each job, with or without SMT, to execute on only one core. However, a task may execute different jobs on different cores.

In a conventional CE scheduler, which does not permit SMT and which requries one frame size across all cores, the primary scheduling decision is how to assign jobs to frames. With CERT-MT, additional decisions are needed: what frame size should be used for each core, on what core should each job be scheduled—with different frame sizes, cores are not interchangeable—and how, if at all, should jobs be co-scheduled? Even so, the requirements given for correctness in Definition 2.7 are unchanged.

However, these correctness requirements are more complicated, particularly since the scheduling decisions needed cannot be made independently. To make them, we employ a mathematical optimization program. To convert our correctness conditions into mathematical form, we define a variable for every possible combination of jobs, cores, and frames.

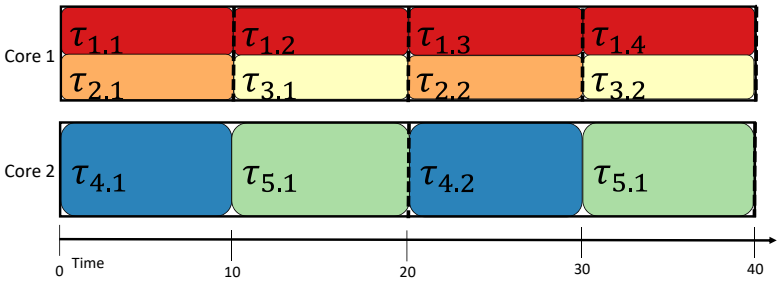

Figure 5.1: A possible schedule and corresponding x variables for the task system of Ex. 5.2.
x(1.1, 2.1, 1, 1)=x(2.1, 1.1, 1, 1)=1
x(1.2, 3.1, 1, 2)=x(3.1, 1.2, 1, 2)=1
x(1.3, 2.2, 1, 3)=x(2.2, 1.3, 1, 3)=1
x(1.4, 3.2, 1, 4)=x(3.2, 1.4, 1, 4)=1
x(4.1, 4.1, 2, 1)=1
x(5.1, 5.1, 2, 1)=0.5
x(4.2, 4.2, 2, 2)=1
x(5.1, 5.1, 2, 2)=0.5

Definition 5.6. For $i \neq j$, let the variable $x(i.a, j.b, \ell, g)$ be defined as

$$x(i.a, j.b, \ell, g) = \frac{\text{time budgeted for } \tau_{i.a:j.b} \text{ on core } \ell \text{ in frame g}}{C^+_{i(j)}}.$$

This actually defines two variables for every pair of distinct jobs: one for which $i < j$ holds, and a second for which the opposite is true. For $i = j$, we define the variable only for $a = b$ and keep the same definition with the provision that a solo job is said to be "simultaneously co-scheduled" with itself.[3] We refer to these variables as the x variables. ◀

Example 5.2. Let the task system τ consist of $\tau_1 = (7.5, 10)$, $\tau_2 = (5, 20)$, $\tau_3 = (5, 20)$, $\tau_4 = (10, 20)$, and $\tau_5 = (20, 40)$. Furthermore, let $C^+_{1(2)}$ and $C^+_{1(3)} = 10$. Figure 5.1 shows a possible schedule for τ on two cores (note that τ has total utilization of 2.25; it cannot be scheduled on two cores without SMT), along with the non-zero corresponding x variables. Frame borders are indicated by dashed lines; we have $\phi(1) = 10$ and $\phi(2) = 20$. The first two variables, $x(1.1, 2.1, 1, 1)$ and $x(2.1, 1.1, 1, 1)$, show that the first jobs of τ_1 and τ_2 are co-scheduled in frame 1 of core 1 for 10 time units, their joint cost. The last variable listed, $x(5.1, 5.1, 2, 2)$, shows that job 1 of τ_5 is scheduled in frame 2 of core 2 for half of its total cost. The preemption of $\tau_{5.1}$ by the scheduler at time 20 is permitted, as our non-preemption requirement only applies to jobs using SMT. ◊

We outline the mathematical constraints needed to fulfill the requirements of Definition 2.7 (i.e., Definition 5.5). These constraints can be used to build an optimization program such that solving the program, i.e., assigning values to the x and frame-size variables so that all constraints hold, produces a correct schedule.

[3] We do not propose scheduling a job with a second copy of itself; we overload the term "simultaneously co-scheduled" to avoid constantly addressing solo jobs as a special case.

Our first constraint addresses the existence of two variables for every pair of jobs. We require that

$$x(i.a, j.b, \ell, g) = x(j.b, i.a, \ell, g) \tag{5.1}$$

holds. In Figure 5.1, we see the first eight variables listed as pairs for this reason. Our remaining constraints fulfill the requirements of Definition 5.5.

(i) **All jobs are scheduled.** To guarantee that all jobs released over the hyperperiod are scheduled, we require that for any job $\tau_{i.a}$ released within the first hyperperiod, the x variables corresponding to it must sum to 1. Mathematically,[4]

$$\forall i, a \sum_{j=i}^{n} \sum_{b=1}^{\frac{H}{T_j}} \sum_{\ell=1}^{m} \sum_{g=1}^{\left\lfloor \frac{H}{\phi(\ell)} \right\rfloor} x(i.a, j.b, \ell, g) = 1.0.$$

In Example 5.2, τ_1 through τ_4 fulfill this requirement by having one variable valued at 1 for each job within the hyperperiod; τ_5 has two non-zero variables for its one job, which is scheduled in two different frames of core 2; each variable is valued at 0.5

(ii) **Any non-preemptable job must be scheduled in exactly one frame.** For $i \neq j$, we require that x equal either zero or one. This requirement can be seen with the variables corresponding to τ_1 through τ_3 in Example5.2. While each job of τ_4 is also scheduled in a single frame, doing so is not necessary for a correct schedule, since τ_4 does not use SMT.

(iii) **Every job completes in a frame that ends no later than its deadline.** Frame g of core ℓ ends at time $g \cdot \phi(\ell)$. We make the per-core frame size into a variable—$\phi(\ell)$ for all $\ell \leq m$—and require that if a job is scheduled in a given frame, then its deadline can be no sooner than the end of the frame. In our optimization program, we express this rule using the following constraint:

$$\forall i.a, j.b, \ell, g, \lceil x(i.a, j.b, \ell, g) \rceil \cdot g \cdot \phi(\ell) \leq d(i.a, j.b).$$

If $\lceil x(i.a, j.b, \ell, g) \rceil = 1$ holds, then the restriction is true if and only if the frame's end time, $g \cdot \phi(\ell)$, is no more than the deadline. If $\lceil x(i.a, j.b, \ell, g) \rceil = 0$ holds, i.e., the job(s) is (are) not scheduled in frame g

[4]Variables for which $i = j$ and $a \neq b$ technically do not exist per Definition 5.6. Here and elsewhere when $i = j$, $\sum_{b=1}^{\frac{H}{T_j}}$ would more properly written as $\sum_{b=a}^{a}$. We leave it as is for the sake of brevity.

of core ℓ, making the frame size irrelevant, then the constraint is always true. While the ceiling operator is meaningless for non-preemptable co-scheduled jobs, which already have $x(i.a, j.b, \ell, g) \in \{0,1\}$, it is necessary for the solo jobs to test whether *any* portion of the job is within frame g.

This constraint holds in Figure 5.1; using the first jobs of τ_1 and τ_2 as an example, we have

$$\lceil x(1.1, 2.1, 1, 1) \rceil \cdot 1 \cdot \phi(1) \leq d(1.1, 2.1)$$
$$1 \cdot 1 \cdot 10 \leq 10.$$

Observe also that scheduling $\tau_{1.1}$ anywhere other than frame 1 of core 1, given that $\phi(2) = 20$, would violate this constraint. In addition, constraint (v) below guarantees that every job can execute for its scheduled time within each frame.

(iv) **No job executes in a frame that begins before its release.** Frame g of core ℓ begins at time $(g-1) \cdot \phi(\ell)$. Consequently, we require that

$$\lceil x(i.a, j.b, \ell, g) \rceil \cdot r(i.a, j.b) \leq (g-1)\phi(\ell)$$

holds. The logic here is similar to that in (iii); if $x(i.a, j.b, \ell, g) = 0$ holds, then the constraint will always be true. Otherwise, the job's release must fall no later than the beginning of the frame for the constraint to hold. In Figure 5.1, using the third job of τ_1 and the second job of τ_2 as examples, we have

$$\lceil x(1.3, 2.2, 1, 2) \rceil \cdot r(1.3, 2.2) \leq (3-1) \cdot \phi(1)$$
$$1 \cdot 20 \leq 2 \cdot 10.$$

(v) **The total execution time scheduled in each frame is no greater than the frame size.** Each x variable requires that $x(i.a, j.b, \ell, g) \cdot C^+_{i(j)}$ units of time be allocated to the corresponding job in the selected frame. We must avoid overscheduling frames. The following constraint accomplishes that goal:

$$\forall\, \ell, g : \sum_{i=1}^{n} \sum_{a=1}^{\frac{H}{T_i}} \sum_{j=i}^{n} \sum_{b=1}^{\frac{H}{T_j}} x(i.a, j.b, \ell, g) \cdot C^+_{i(j)} \leq \phi(\ell).$$

For example, $C^{+}_{1(2)}$ and $f(1)$ are equal to 10 in Example 5.2, giving us

$$x(1.1, 2.1, 1, 1) \cdot C^{+}_{1(2)} \leq \phi(1)$$
$$1 \cdot 10 \leq 10$$

for frame 1 of core 1; note that all other variables connected to this frame, apart from $x(2.1, 1.1, 1, 1)$, are equal to 0.

(vi) **No job executes in parallel with itself.** For non-preemptable jobs, this rule holds automatically, since every job is scheduled in exactly one frame. For preemptable solo jobs, we add the constraint

$$\forall i, a, \ell \sum_{g=1}^{\left\lfloor \frac{H}{\phi(\ell)} \right\rfloor} x(i.a, i.a, \ell, g) \in \{0, 1\},$$

which states that for every solo job, either all or none of it must be on a given core. In Example 5.2, this constrains each job of τ_4 and τ_5 to a single core.

If a correct schedule exists, then an optimization program that follows the above constraints can find one. Despite the name, there is no objective function that needs to be optimized; if a set of decision variables that fulfills all restrictions exists, a correct schedule can be formed by making the scheduling choices corresponding to those variables.

5.2 CERT-MT Schedulability Studies

To evaluate our work, we conducted a large scale synthetic-task schedulability study. As in Chapter 4, we define scenarios by a per-task utilization range, core count, and SMT parameters and summarize each scenario in a single graph. Parameters are described in Section 5.2.1 below.

For each scenario, we created synthetic systems with total utilizations ranging from $\frac{3m}{4}$ to $2m$. In preliminary observations, we found that not all task systems with utilization m could be scheduled on m cores, but all task systems with utilization $\frac{3m}{4}$ could be. We calculated RSAs on the assumption that all systems with utilization less than $\frac{3m}{4}$ were schedulable. We plotted graphs showing the percentage of systems schedulable at each utilization level in increments of 0.25. Each point on our graphs corresponds to at least

50 task systems. We use larger increments and smaller sample sizes than we did in Chapter 4 due to the increased time required to test each task system.

An ideal scheduler without SMT but with the ability to preempt and migrate jobs without restrictions would have an RSA of 1.0; it could schedule all task systems with total utilization at most m and no task systems with greater utilization. However, many HRT schedulers fall far short of this mark. For any multicore platform, there exist HRT task systems with utilization arbitrarily close to one that cannot be scheduled using G-EDF (Dhall and Liu, 1978); this is known as the Dhall Effect and was described in Section 2.3.3. As another example of HRT schedulers being less than ideal, under partitioned scheduling a system of n tasks each with $u_i > 0.5$ requires n cores to schedule (see Section 2.3.3), potentially causing RSA to be as low as $0.5 + \varepsilon$. A detailed look at the limitations of HRT schedulers can be found in (Carpenter et al., 2004).

5.2.1 Schedulability Test Parameters

We use five parameters, summarized in Table 5.2, to define each scenario. The first parameter, core count (m), specifies the number of cores in our target platform. Possible values are four and eight; due to the complexity of our scheduling problem, we were unable to conduct sixteen-core schedulability studies.

The second parameter again determines task utilizations before accounting for SMT. We use the light, medium, wide, and heavy per-task utilization ranges, defined identically to their use in Section 4.3.

Whereas we only needed to know utilizations for SRT scheduling, task costs and periods are relevant in this case. Each task had a period randomly selected from the set $\{10, 20, 40, 80\}$, with each possible period being equally likely. Baseline costs without SMT were computed as the product of period and utilization.

The remaining three parameters—$E[f_i(1)]$, slope, and distribution—govern the effects of SMT on execution time. They are slightly different from those used in Chapter 4. $E[f_i(1)]$ gives the overall expected value for $M_{i(j)}^{(1)}$ *given that* $C_i \leq C_j$ holds. Possible values are 0.35 (optimistic), 0.55 (mid-range), and 0.75 (pessimistic). For each task τ_i, we set $f_i(1)$ (see Definition 3.19) equal to a random exponential variable with mean $E[f_i(1)]$. The *slope* parameter is used in conjunction with $f_i(1)$ to define the linear regression function $f_i\left(\min\left(\frac{C_i}{C_j}, 1\right)\right)$. Possible values are 0.0, 0.15, and 0.30.

Finally, the *distribution* parameter determines whether $M_{i(j)}^{(1)}$ is equal to $f_i\left(\min\left(\frac{C_i}{C_j}, 1\right)\right)$—the *fixed* distribution—or equal to an exponential random variable with mean $f_i\left(\min\left(\frac{C_i}{C_j}, 1\right)\right)$; the exponential distribution, abbreviated as *expo.* Parameter choices are summarized in Table 5.2. The parameters used for

Table 5.2: Summary of parameters used to define schedulability scenarios

Parameter description	Parameter symbol	Possible Values
core count	m	4, 8
per-task util. distribution	none	U(0, 0.4) (light); U(0.3, 0.7) (medium); U(0, 1) (wide); U(0.6, 1.0) (heavy)
expected $M_{i(j)}^{(1)}$ given $C_i \leq C_j$	$E[f(1)]$	0.2, (optimistic); 0.4, (mid-range); 0.6 (pessimistic)
slope of regression function	none	0.0, 0.15, 0.30
distribution for $M_{i(j)}^{(1)}$ given f_i	none	fixed, exponential

each graph are given in the graph's title. In the graph titles, we write $f(1)$ rather than $E[f(1)]$ to improve readability.

In addition, we specified a timeout value for each scenario: if a feasible solution for a system could not be found within the timeout, we counted the system as being unschedulable. For most of our scenarios, we used a 60-second timeout. Because we observed many systems deemed unschedulable due to timeouts rather than true impossibility, we repeated a subset of our experiments using a 300-second timeout.

5.2.2 Schedulability Test Results: Four Cores and 60 Second Timeouts

We tested 72 four-core systems using all combinations of task system parameters and a 60-second timeout. We discuss our observations from those scenarios here. The graphs depicted here are chosen to illustrate the trends we observed; complete graphs are available in Appendix B. Results using a 300-second timeout and results for eight-core systems are discussed in Sections 5.2.3 and 5.2.4.

Observation 5.1. *In the best case, CERT-MT produced an RSA of 1.22.* The best-case scenario is shown in Figure 5.2. We were able to do better when allowing a 300-second timeout, getting an RSA as high as 1.34.

Observation 5.2. *In the worst case, CERT-MT produced an RSA of 0.95.* Figure 5.3 shows our worst scenario. While this result falls short of an ideal scheduler without SMT, it does outperform many commonly-used schedulers, such as P-EDF.

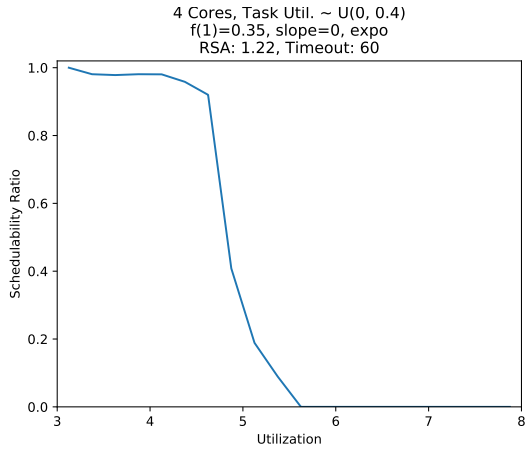

Figure 5.2: The best scenario.

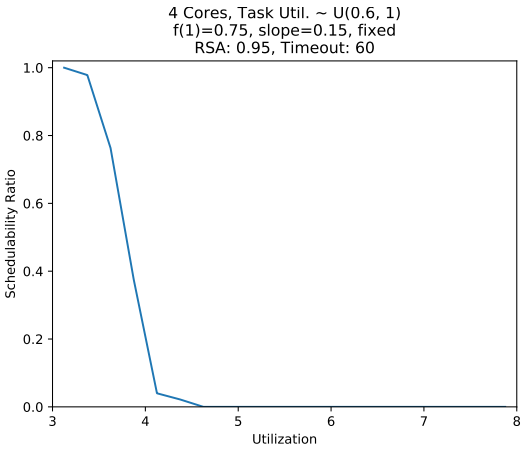

Figure 5.3: The worst scenario.

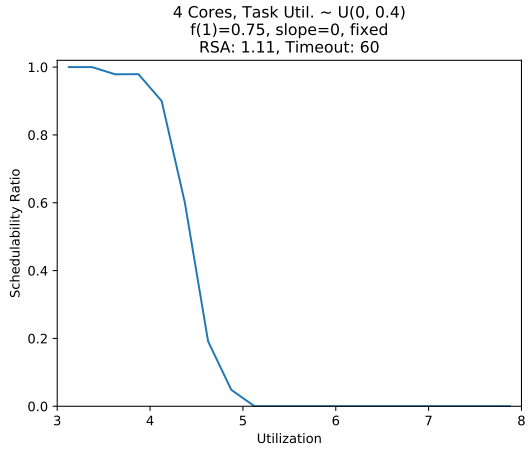

Figure 5.4: The median scenario.

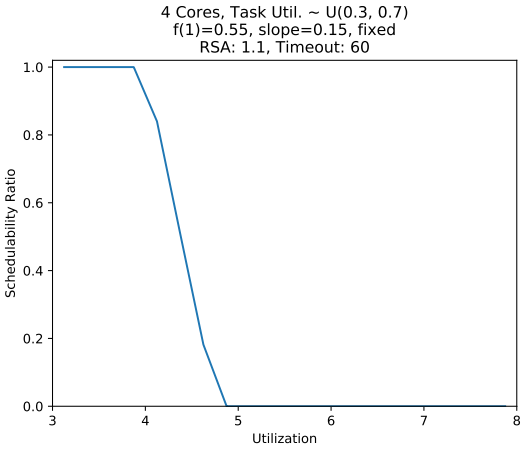

Figure 5.5: A mean scenario.

Observation 5.3. *CERT-MT was beneficial even compared to an ideal scheduler in more than half of scenarios tested*, with a median RSA of 1.11. Figure 5.4 depicts our median scenario and Figure 5.5 depicts a mean scenario.

Observation 5.4. *CERT-MT performed the best given the light per-task utilization ranges and saw the poorest performance given heavy per-task utilization.* The per-task distribution range was the single most important factor influencing RSA. Our best result (Figure 5.2) uses light per-task utilization and our worst result (Figure 5.3) uses heavy per-task utilization. Figures 5.6 through 5.9 show scenarios with the same SMT parameters but different per-task utilizations. Figure 5.6, depicting light per-task utilization, has an RSA of

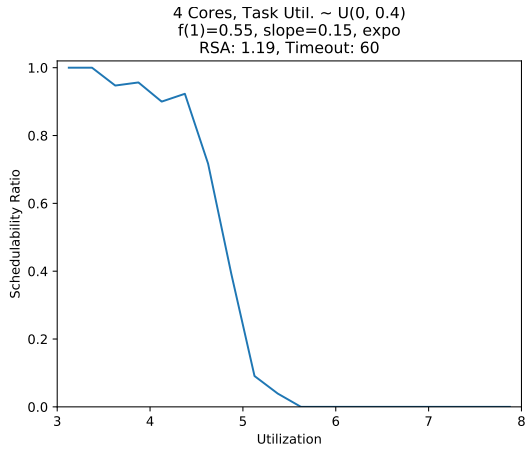

Figure 5.6: Light per-task utilization.

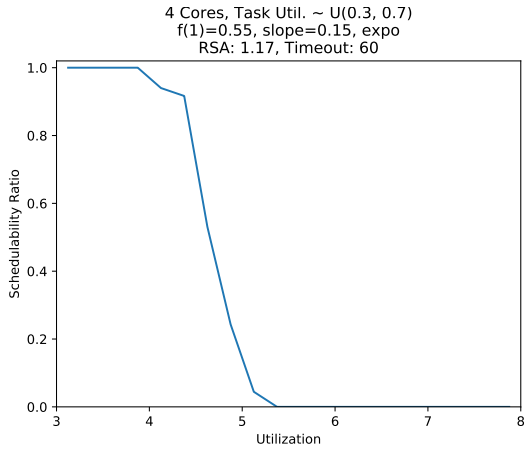

Figure 5.7: Moderate per-task utilization.

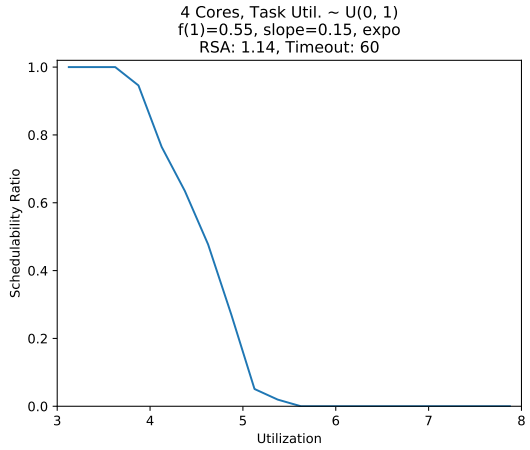

Figure 5.8: Wide per-task utilization.

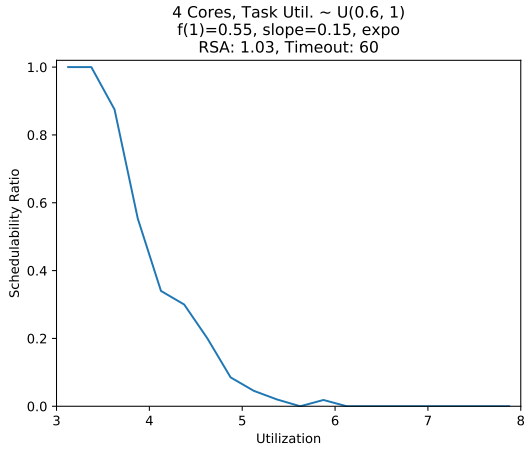

Figure 5.9: Heavy per-task utilization.

1.19, whereas Figure 5.9 has an RSA of 1.03. This effect is similar to what we saw in the SRT case; tasks with baselines utilizations close to 1.0 are less likely be to be able to take advantage of SMT.

Observation 5.5. *Scenarios using the exponential distribution tended to perform better than those using the fixed distribution.* This is the opposite of what we say saw for the SRT case in Chapter 4. For example, Figures 5.5 and 5.15 differ only in that Figure 5.5 uses the fixed distribution and Figure 5.15 uses the exponential distribution, but Figure 5.5 has an RSA of 1.1, whereas Figure 5.15 has an RSA of 1.17. Note also that our best-case scenario (Figure 5.2) uses the exponential distribution while our worst (Figure 5.3) uses the fixed distribution. In the SRT case, every task using SMT was assumed to affect every other task using

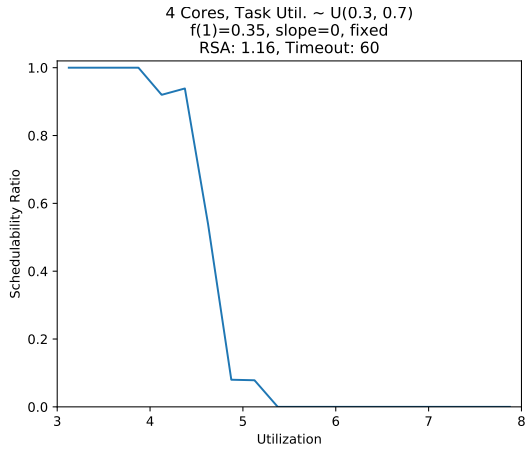

Figure 5.10: $E[f(1)] = 0.35$, slope 0.

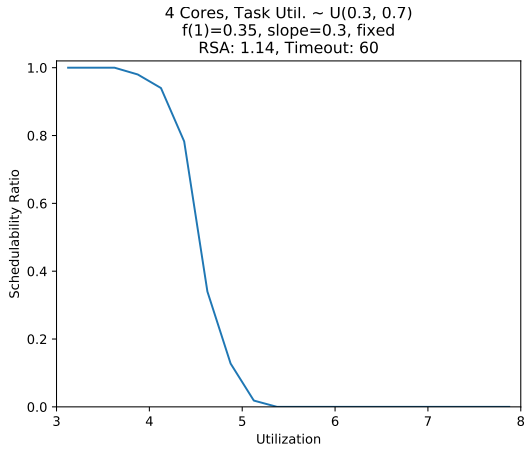

Figure 5.11: $E[f(1)] = 0.35$, slope 0.3.

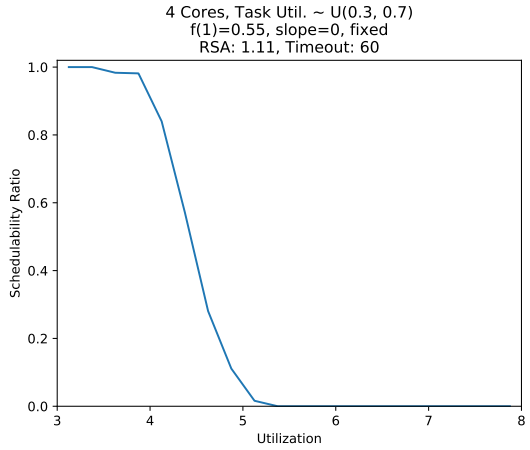

Figure 5.12: $E[f(1)] = 0.55$, slope 0.

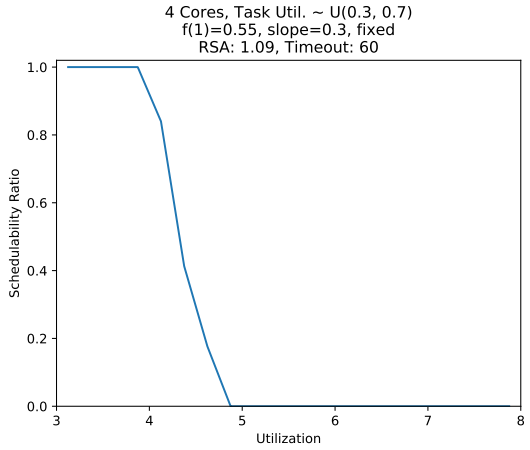

Figure 5.13: $E[f(1)] = 0.55$, slope 0.3.

SMT; thus any parameter causing additional variation tended to be harmful. Here, that effect is mitigated by pairing tasks individually.

Observation 5.6. *Changing the slope of f_i generally had a smaller impact than changing the expected value of $f_i(1)$.* For example, see Figures 5.10 through 5.13. When comparing Figure 5.10 to Figure 5.12, RSA decreases from 1.16 to 1.11, a drop of 0.05. When comparing Figure 5.11 to 5.13, RSA drops again drops by 0.05, from 1.14 to 1.09. In both of these comparisons, $E[f(1)]$ changes as slope remains constant.

The decrease in RSA is less when comparing graphs with the same $f(1)$ values but different slopes, i.e., comparing graphs on the left and right as Figures 5.10 through 5.13 are arragned. Between both left-right pairs, RSA decreases by only 0.02.

Observation 5.7. *One-minute timeouts were a significant issue on four-core systems, particularly with light per-task utilizations.*

For each of our 72 scenarios, we counted how many task systems were deemed unschedulable due to timeouts and how many were "true failures" where the system was actually shown to be infeasible. In 21 scenarios, all failures were due to timeouts rather than determining the system was not feasible; in an additional 33 scenarios, timeout failures outnumbered true failures. True failures outnumbered timeout failures only when wide per-task utilization was used. These scenarios would also have the fewest tasks. Given the frequency of timeouts, it seems likely that our timeout requirement caused CERT-MT results to be understated for all systems using the moderate, wide, and light per-task utilizations. To be clear, scenarios were evaluated using a shared research cluster; we had no control over the exact schedule or quality of service. For this reason, our observations on the time required to test individual task systems are useful mainly for order-of-magnitude comparisons.

5.2.3 Schedulability Test Results: Four Cores and 300 Second Timeouts

We repeated our schedulability studies with a 300-second timeout for eighteen four-core scenarios: all four per-task utilization distributions, all three possible values for $E[f_i(1)]$, a slope of 0.15, and both the fixed and exponential distributions. We summarize the effects of increasing the timeout for each of the four utilization descriptions.

Definition 5.7. When comparing two scenarios with different time values and otherwise identical parameters, let the *timing-related change* be defined as the RSA of the scenario with the longer timeout value—here 300 seconds—divide by the RSA of scenario with the shorter timeout value. ◄

A value of 1.0 for timing-related change indicates no change; a value greater than 1.0 indicates that increasing the timeout allowed for otherwise unschedulable systems to be schedulable.

For the heavy, moderate, and wide distributions, increasing the timeout had minimal effect; timing related changes across all three distributions ranged from 0.98 to 1.02, with means of 1.00 for the heavy and wide cases and 1.01 for the moderate case. For the heavy scenarios, two had no timeout-induced scheduling failures and four had one each. For the wide scenarios, timeout-induced failures outnumbered true failures by an approximately two to one margin. For the moderate scenarios, timeout failures outnumbered true failures by as much as 100 to 1.

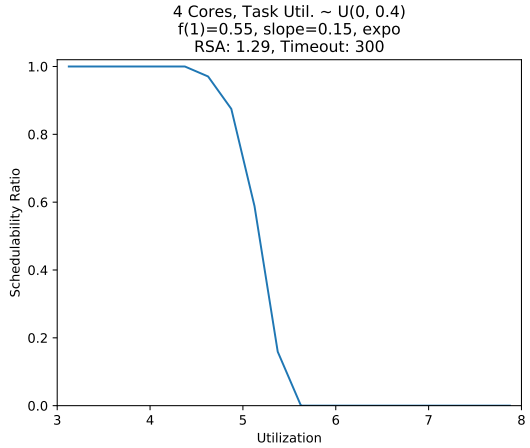

Figure 5.14: Light per-task utilization, 300-second timeout.

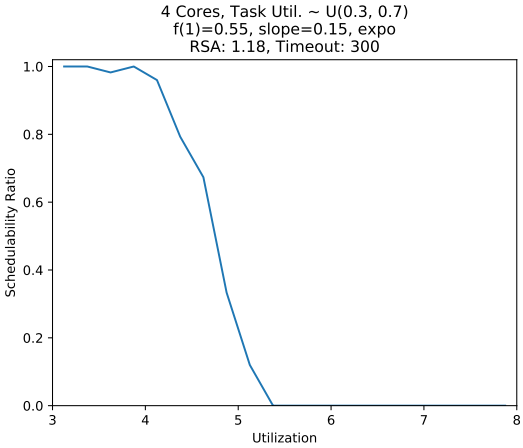

Figure 5.15: Moderate per-task utilization, 300-second timeout.

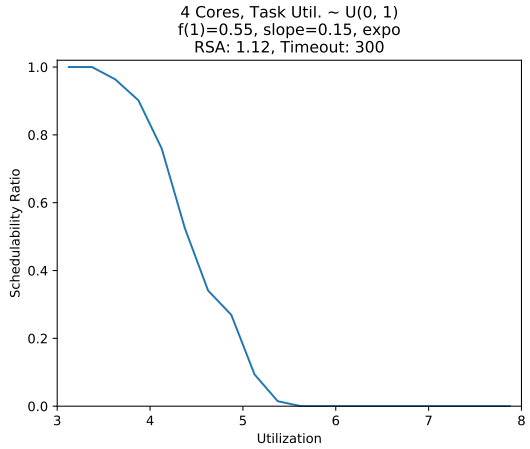

Figure 5.16: Wide per-task utilization, 300-second timeout.

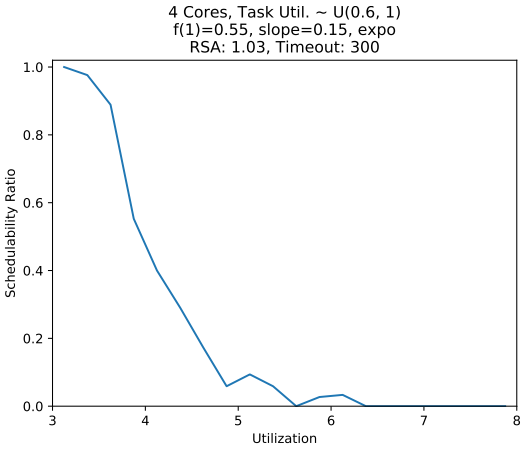

Figure 5.17: Heavy per-task utilization, 300-second timeout.

For the light scenarios, we continued to see no true failures. However, the longer timeout did allow for significantly better schedulability; values on timing related changes ranged from 1.07 to 1.11, with a mean of 1.08. RSA values ranged from 1.15 to 1.34, with a mean of 1.25. Among all 60-second scenarios, the maximum RSA was 1.22.

Figures 5.14 through 5.17 show the scenarios for $E[f_i(1)] = 0.55$, the exponential method, and all four per-task utilization ranges; apart from the changed timeout value, these scenarios are the same as those of Figures 5.6 through 5.9.

5.2.4 Schedulability Test Results: Eight Cores

We ran schedulability studies for eight eight-core scenarios: all four per-task utilization distributions, $E[f_i(1)] = 0.55$, a slope of 0.15, and both the fixed and exponential distributions.

When using light per-task utilization with eight cores and a 60-second timeout, almost every attempt to find a schedule timed out.

For the wide and moderate distributions, RSA was slightly lower in eight-core scenarios. For the heavy distribution, RSA stayed the same when using the fixed distribution but increased slightly when using the exponential distribution.

When we repeated the eight-core experiments with a 300-second timeout, RSAs for the moderate, wide, and heavy eight-core scenario were equal to or slightly greater the corresponding scenario with four cores and a 60-second timeout, but less than the corresponding four-core scenario with a 300-second timeout. The light scenario saw its RSA improve to 1.02 with a 300-second timeout; much better than the 60-second case, but still inferior to both the four-core scenario and eight-core scenarios with different per-task utilizations.

Figures 5.18 through 5.21 show the scenarios for $E[f_i(1)] = 0.55$, the exponential method, all four per-task utilization ranges, and a 60-second timeout. apart from the changed core count, these scenarios are the same as those of Figures 5.6 through 5.9. Figures 5.22 through 5.25 show the same eight-core scenarios with a 300-second timeout.

5.3 A More Practical Approach to SMT-Aware Scheduling

In this section, we give an SMT-aware HRT scheduler designed to address two key weaknesses of CERT-MT: scalability problems and the use of CE scheduling. The disadvantages of poor scalability, as

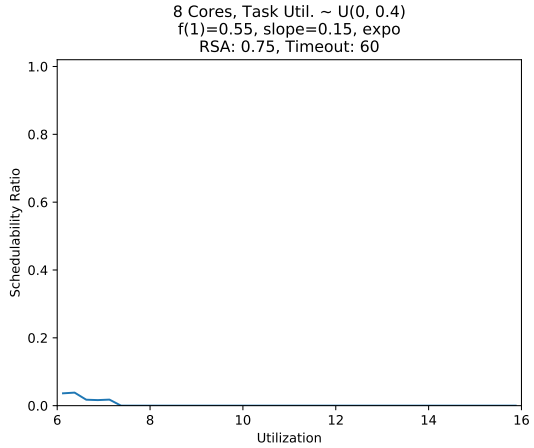

Figure 5.18: Light per-task utilization, eight cores, 60-second timeout.

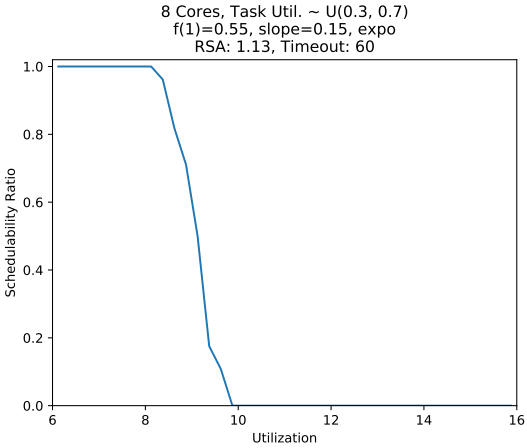

Figure 5.19: Moderate per-task utilization, eight cores, 60-second timeout.

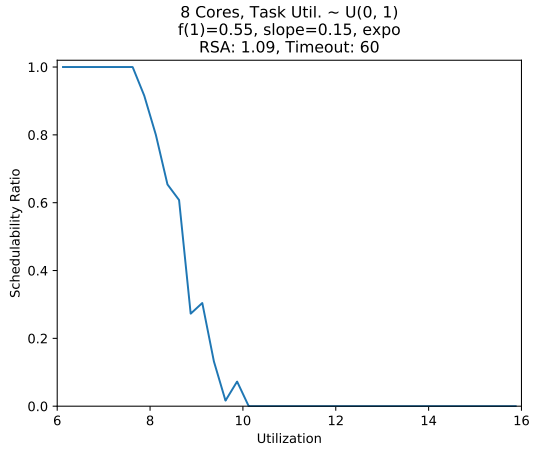

Figure 5.20: Wide per-task utilization, eight cores, 60-second timeout.

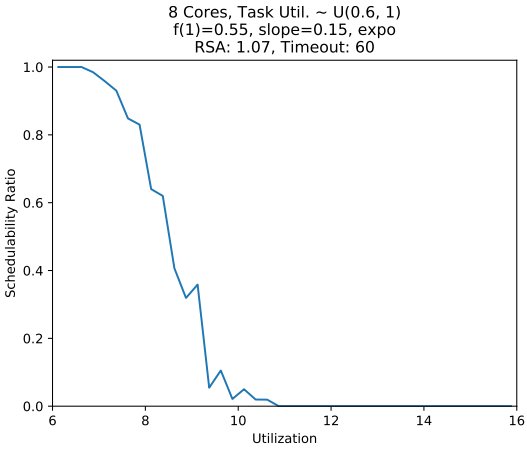

Figure 5.21: Heavy per-task utilization, eight cores, 60-second timeout.

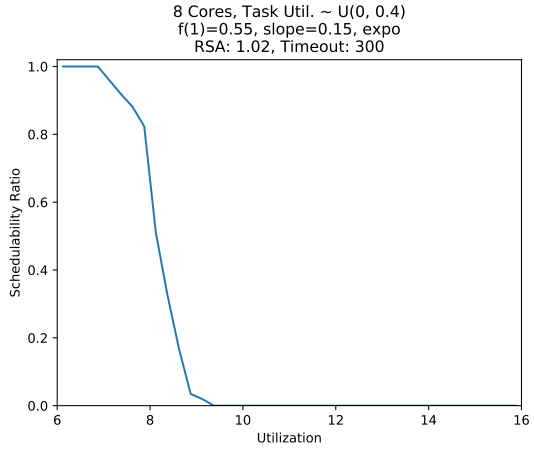

Figure 5.22: Light per-task utilization, eight cores, 300-second timeout.

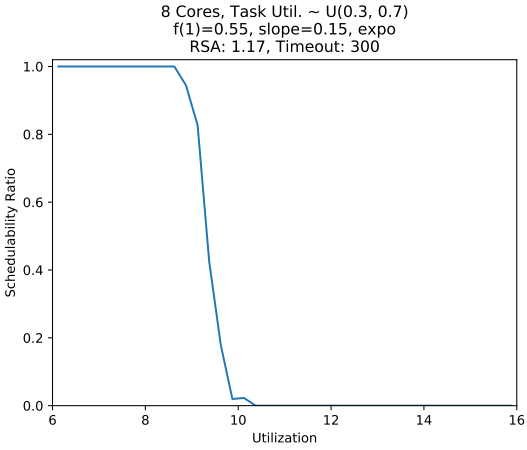

Figure 5.23: Moderate per-task utilization, eight cores, 300-second timeout.

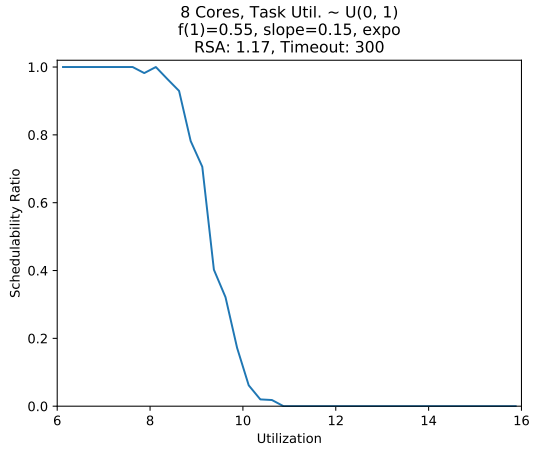

Figure 5.24: Wide per-task utilization, eight cores, 300-second timeout.

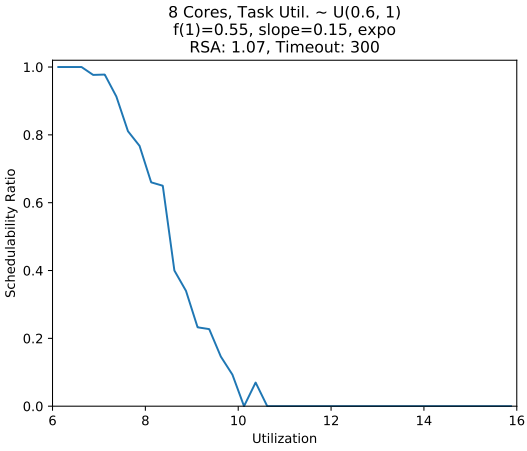

Figure 5.25: Heavy per-task utilization, eight cores, 300-second timeout.

seen in Section 5.2.4, are obvious. Disadvantages of CE scheduling include the need to recalculate the entire schedule should even one task change and the fact that many existing systems use *priority-driven scheduling*, in which each job has a priority and scheduling consists of executing jobs with the highest priority first. EDF is as an example of priority-driven scheduling. Another example is *rate-monotonic* (RM) scheduling, in which highest priority goes to the job belonging to the task with the shortest period. For existing systems using priority-driven approaches, implementing SMT-aware scheduling may be easier if the priority-driven framework can be maintained, possibly making SMT a more attractive option.

Our priority-driven scheduling method consists of three steps, which are depicted graphically in Figure 5.26. First, we use an optimization program to tranform a task system τ into a system τ^R that uses SMT.[5] The decision to use SMT is made on a per-task basis. This choice is fundamental to the rest of our process, as SMT usage affects many aspects of task behavior. Second, we partition τ^R into subsystems that are assigned to individual cores, with subsystem $\tau^{R\ell}$ denoting the set of tasks assigned to core ℓ. Third, we schedule subsystems on each core using a priority-driven method such as EDF scheduling.

Even for the largest systems tested, our transformation step consistently requires less than three seconds to perform; this stage of testing potential systems will not become a bottleneck in the development process. The cost of our quick transformation is that our methods only apply to systems in which many tasks share common periods. However, this restriction is an easy price to pay, given that industrial-oriented task systems are commonly built around only a few distinct periods. We evaluate this approach in Section 5.4 via a large-scale schedulability study in which we track both the proportion of task systems that can be scheduled on a given platform and how much time is needed to conduct each test.

5.3.1 Transforming the System

In this subsection we show how to transform τ into an equivalent system τ^R in which some tasks are replaced by *paired* tasks.

Definition 5.8. Two tasks τ_i and τ_j are *paired* if a decision has been made that all jobs of τ_i will be simultaneously co-scheduled (Definition 3.7) with a job of τ_j and all jobs of τ_j will be co-scheduled with a job of τ_i. τ_i and τ_j can be paired only if they share a common period. The paired task is treated as a single schedulable unit $\tau_{i:j}$ with cost $C^+_{i(j)}$ (Definition 5.3). ◄

[5]We use the "R" superscript to avoid any confusion with T for time.

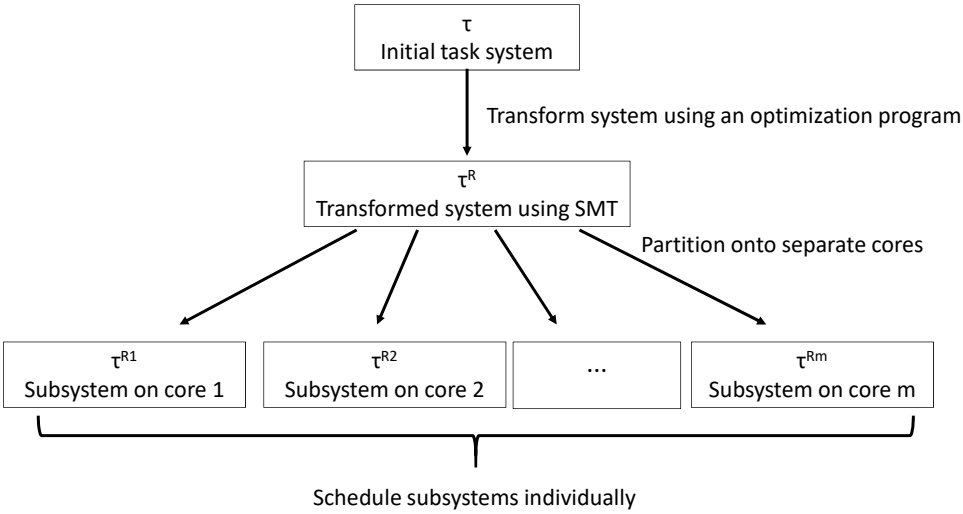

Figure 5.26: Overview of the Priority-Driven approach to SMT scheduling.

"Equivalent" here means that if τ^R is scheduled correctly, then τ is also scheduled correctly. Because we require that paired tasks share a period, a paired task $\tau_{i:j}$ has the same relative deadline as its component tasks τ_i and τ_j. Consequently, if $\tau_{i:j}$ is scheduled correctly, then τ_i and τ_j are also scheduled correctly. It follows that τ can be correctly scheduled by combining some tasks into pairs and then correctly scheduling all solo tasks and all task pairs, treating each task pair as if it were a single task.

To aid in our explanations, we define a system's total utilization when task pairs are treated as if they were individual tasks.

Definition 5.9. The *transformed utilization* U^R of system τ^R is given by

$$\sum_{\forall i:\tau_i \text{ is a solo task}} u_i + \sum_{\forall i,j:i>j,\tau_i \text{ and } \tau_j \text{ are paired}} \frac{C^+_{i(j)}}{T_i}. \tag{5.2}$$

We use $U^{R\ell}$ for the equivalent term when considering only tasks and task-pairs assigned to a single core ℓ. ◄

5.3.1.1 Which Tasks Should be Paired?

Given the role that total utilization plays in determining schedulability, it is reasonable to define task pairs so as to minimize total paired utilization. We can do so using an optimization program with decision variables $x_{i:j}$ for all tasks τ_i and τ_j within τ.

Definition 5.10. For all i and all j such that $T_i = T_j$, let $x_{i:j}$ equal 1 if τ_i and τ_j are paired with each other and 0 otherwise. For $i = j$, let $x_{i:j}$ equal 1 if τ_i is a solo task in τ^R and 0 otherwise. Since we do not consider pairing tasks where $T_i \neq T_j$, we define $x_{i:j} = 0$ for those cases. ◄

With Definition 5.10 in place we can write U^R as follows:

$$\sum_{i=1}^{n} \sum_{j=i}^{n} x_{i:j} \cdot \frac{C^+_{i(j)}}{T_i}. \tag{5.3}$$

Recall from Definition 5.3 that for solo tasks, we define $C^+_{i(j)} = C_i$; hence for $i = j$, $\frac{C^+_{i(j)}}{T_i} = u_i$.

5.3.1.2 Optimization Program Constraints

In order for τ^R to be equivalent to τ, all tasks within τ must be accounted for in τ^R. To enforce this rule, we require that

$$\forall i \leq n : \sum_{j=1}^{n} x_{i:j} = 1 \tag{5.4}$$

holds; essentially, all tasks within τ must appear in τ^R either as a solo task or as part of a paired task.

Additionally, just as τ will be unschedulable under our assumption of implicit deadlines if $C_i > T_i$ holds for any task, τ^R will be unschedulable if $C^+_{i(j)} > T_i$ holds for any paired task $\tau_{i:j}$. We therefore require that the following holds:

$$x_{i:j} \cdot C^+_{i(j)} \leq T_i, \tag{5.5}$$

i.e., τ_i and τ_j may be paired only if $C^+_{i(j)} \leq T_i$ holds. Since we require that only tasks sharing a period may be paired, we do not need a separate restriction governing the relative values of $C^+_{i(j)}$ and T_j.

Finally, note that Definition 5.10 actually defines both $x_{i:j}$ and $x_{j:i}$ for each possible task pair. To avoid any inconsistencies, we add the restriction that

$$\forall i, j : x_{i:j} = x_{j:i}. \tag{5.6}$$

We define our optimization program as an integer linear program minimizing Exp. (5.3) subject to Exps. (5.4) through (5.6). Although integer linear programming has exponential complexity relative to the number of variables, our program executed quickly in the experiments presented in Sec. 5.4. We discuss execution times in more detail as part of our experimental evaluation.

5.3.2 Partitioning the Transformed System

After defining τ^R, our next step is to partition it onto the cores of π. Even without non-preemptive sections, assigning tasks and task pairs to cores so that all cores are schedulable is a bin-packing problem. While bin-packing is NP-complete in the strong sense, multiple well-studied approximation algorithms for it exist. We use two of these algorithms—worst-fit decreasing and best-fit decreasing bin-packing—and two algorithm variations of our own, giving us a total of four partitioning algorithms. In all cases, we assign tasks to cores in non-increasing order of $\frac{C^+_{i(j)}}{T_i}$ and view each core as a single bin with capacity 1.0.

Worst-fit decreasing and best-fit decreasing. In worst-fit bin packing, each task or paired task is placed on the core that will maximize remaining capacity on the selected core. In best-fit bin packing, each task or paired task is placed on the core that will minimize remaining capacity on the selected core.

Period-aware bin-packing. In our second two algorithms, we modify the worst-fit and best-fit algorithms in an attempt to limit the number of different periods on any one core. Having all tasks on a core share a common period eliminates the ability of non-preemptable tasks to block higher-priority tasks; one consequence of Definition 2.8 and Theorem 2.2 is that if all tasks on a given core share the same period, then no task is subject to priority-inversion blocking. In this case, the core is schedulable if and only if $U^{R\ell} \leq 1$ holds.

In *period-aware worst-fit partitioning,* we again attempt to place tasks and task-pairs on cores in non-increasing order of $\frac{C^+_{i(j)}}{T_i}$. In this method, we potentially make two attempts to assign each task to a core. In the first attempt, we use worst-fit bin-packing to assign a task or paired task to a core, but we consider only cores on which all previously assigned tasks have the same period as the current task. If a task or paired task is assigned to a core at this point, we move on to the next task or paired task. If the task or pair cannot be placed onto a core using this method, we consider all cores of the platform and assign the task using the standard worst-fit decision process.

Period-aware best-fit partitioning is similar, but uses the best-fit approach instead.

5.3.3 Testing Individual Cores

Our final step is to test each core for schedulability. To do so, we treat each paired task as if it were a single task. After tasks have been partitioned, the process is no different from uniprocessor scheduling without SMT. While we use EDF in this paper, there is no reason why another uniprocessor scheduling algorithm,

such as RM, cannot be used; the only change needed to use a different per-core scheduling algorithm would be to use a different schedulability test.

We consider three approaches to task preemption, described below. Strictly speaking, the timing analysis of Chapter 3 only allows for non-preemptive scheduling. However, experimenting with preemption can guide future work to enable safe timing analysis for more cases.

No preemption. In this model, our first and most pessimistic, we make every paired task non-preemptable. This is the model used for CERT-MT in Sections 5.1 and 5.2. In this case, τ_i may be blocked by any task pair $\tau_{j:k}$ for which $T_j > T_i$ holds for time $C^+_{j(k)}$.

Limited preemption. Our next approach allows paired tasks to be preempted after one job has completed, i.e., SMT is not in use at the moment of preemption. To define when we can preempt, we need an additional cost definition. We define the *inner cost* of a paired task as the lesser of the two tasks' execution times, i.e., the time during which SMT is actually in use.

Definition 5.11. The *inner cost* to simultaneously execute jobs of τ_i and τ_j is given by $C^-_{i(j)}$, defined as the execution time for both jobs assuming they begin simultaneously, i.e.,

$$C^-_{i(j)} = \min(C^{(1)}_{i(j)}, C^{(1)}_{j(i)}).$$

If $i = j$, then $C^-_{i(j)} = C_i$, indicating solo execution for τ_i. ◀

In the limited preemption model, we make each paired task non-preemptable during the first $C^-_{i(j)}$ units of their execution, but fully preemptable afterwards. In this case, τ_i may still be blocked by any task pair $\tau_{j:k}$ on the same core for which $T_j > T_i$ holds, but but blocking time can be at most $C^-_{j(k)}$.

Full preemption. Here we assumed that all tasks are fully preemptable, even when there are paired jobs running at the same time. In practice, allowing unrestricted preemptions along with SMT would tend to make timing analysis even harder, possibly making it impossible to guarantee a safe timing analysis for HRT tasks. However, testing this approach allowed us to see the cost of limiting preemptions. In addition, this approach may be viable for SRT and non-safety-critical systems, where some additional uncertainty in timing analysis may be tolerable. With full preemption, τ_i is never blocked; it follows that $b_i = 0$.

In all three cases, schedulability can be tested by applying Theorem 2.2 to test the whether the subsystems assigned to each core can be scheduled using EDF.[6]

[6]Jeffay et al. (Jeffay et al., 1991) provide a tighter, but more complex, test.

5.4 Priority Driven Schedulability Studies

The parameters for our schedulability tests were almost identical to those described in Section 5.2. There were two differences. First, we considered core counts of four, eight, and sixteen instead of four and eight. Second, we differentiate between four period and eight period scenarios. In four period scenarios, as in Section 5.2, each task had a period randomly selected from the set $\{10, 20, 40, 80\}$, with each possible period being equally likely. In eight period scenarios, periods were selected from the set $\{5, 10, 20, 40, 80, 160, 320, 640\}$, again with equal probability.

We tested scenarios for all possible parameter combinations, giving us 216 total graphs; the full set of graphs is in Appendix C.

For each scenario, we give four schedulability curves. Three of them, *no preemption*, *limited preemption,* and *full preemption* were described in Section 5.3.3. The fourth, *baseline*, shows schedulabilty for a fully preemptive partitioned EDF scheduler with partitioning done using worst-fit bin-packing. Each of the three curves other than baseline shows the results of the partitioning algorithm—best fit, worst fit, period-aware best fit, or period-aware worst fit—that gave the best results. Since all four partitioning algorithms execute quickly, it is entirely practical to run all four for each task system and then choose the best result.

For each scenario, we created synthetic systems with total utilizations ranging from $\frac{m}{2}$ to $2m$. We used $\frac{m}{2}$ as a starting point to ensure that our baseline scenario would consistently be able to schedule all task systems. We calculated RSAs on the assumption that all systems with utilization less than $\frac{m}{2}$ were schedulable. Our graphs show the percentage of systems schedulable at each utilization level in increments of 0.25. Each point on our graphs corresponds at least 500 task systems.

We use two metrics to summarize the proportion of systems that are schedulable under each scenario. We use RSA, as before, but a second metric, *relative improvement.*

Definition 5.12. Let the *relative improvement* (RI) of a given scheduling algorithm be equal to its RSA divided by the RSA of a specified baseline algorithm. ◄

We use RI to show the benefit of our methods in cases where partitioned scheduling without SMT falls well short of an ideal scheduler to begin with.

Observation 5.8. *In the best scenario, RSA and RI were both greater than 1.7 with full preemption and greater than 1.55 with no or partial preemption.* Figure 5.27 shows the best scenario for all three preemption models, under either metric.

Observation 5.9. *With partial preemption or no preemption, the worst cases for SMT occurred when the number of periods was greater than the number of cores and the best cases when cores outnumbered periods.* Figure 5.28 shows the worst scenario for both the partial and no preemption models. The scenario considers scheduling task systems with eight periods on a four-core system; it is impossible to partition tasks so that only one period is represented on each core. Consequently, tasks would be subject to extensive blocking, making the system unschedulable.

At first glance, the parameters of Figure 5.28 seem very favorable for SMT; it uses the optimistic μ value, has a slope of zero, and uses the light per-task utilization distribution, which in both Sections 4.3 and 5.2 appeared beneficial to SMT. These parameters do cause excellent performance when SMT is fully preemptable, but when SMT is not preemptable, "optimistic" parameters that allow all tasks to use SMT have the side effect of making all tasks non-preemptable. Figure 5.27 shows the reverse case: cores outnumber periods, meaning that the effects of non-preemption can be mitigated by limiting the number of periods represented on each core. In practice, we would never use SMT when it decreases overall schedulability, i.e., RI is less than one. The best way to use SMT in that case would be to not use it at all.

Observation 5.10. *For all three preemption models, SMT improved schedulability in more than half of tested cases.* Figures 5.31 and 5.32 depict scenarios with RIs of 1.18 (the median for full preemption) and 1.07 (the median for no and partial preemption median).

Observation 5.11. *Heavy per-task utilization makes SMT less effective under full preemption, but may not change the effectiveness of SMT given partial or no preemption.* Figures 5.33 through 5.36 depict scenarios using all four utilization ranges and otherwise identical parameters. For the full preemption model, RSA drops as average per-task utilization increases. For the partial and no preemption models, RSA is highest given light per-task utilization, but is otherwise almost constant.

As per-task utilization increase, baseline schedulabilty using partitioned EDF declines as well. Consequently, the no preemption and partial preemption models have equal RIs given either light or heavy per-task utilization.

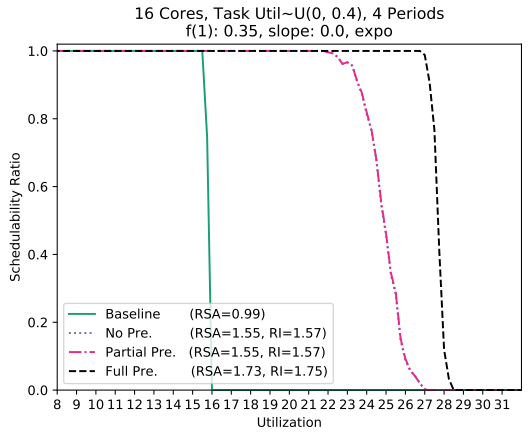

Figure 5.27: The best scenario under all preemption models.

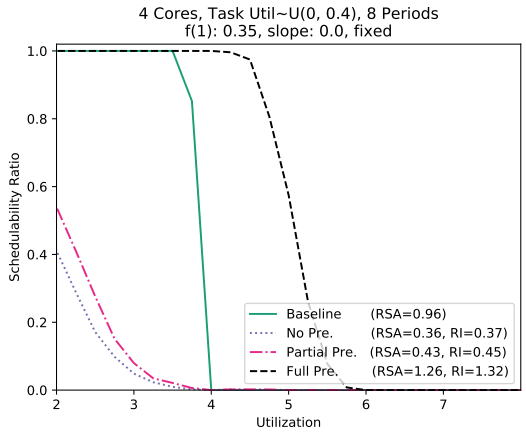

Figure 5.28: The worst scenario with partial or no preemption.

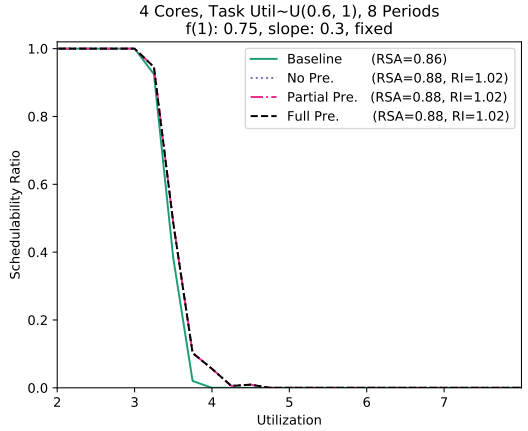

Figure 5.29: The worst scenario by RI for full preemption.

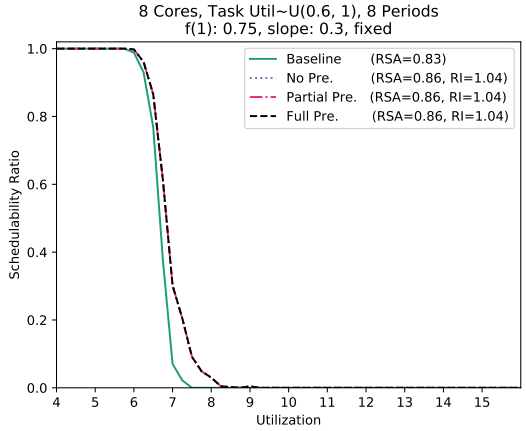

Figure 5.30: The worst scenario by RSA for full preemption.

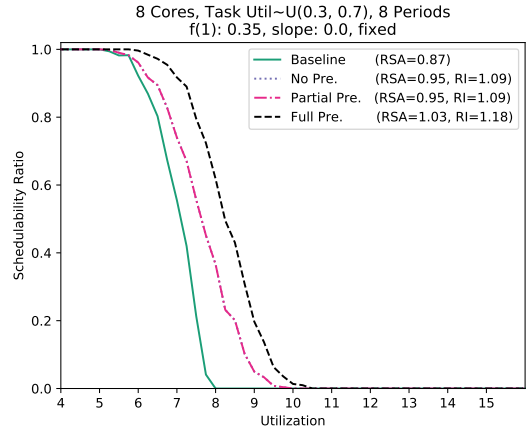

Figure 5.31: A median scenario by RI for full preemption.

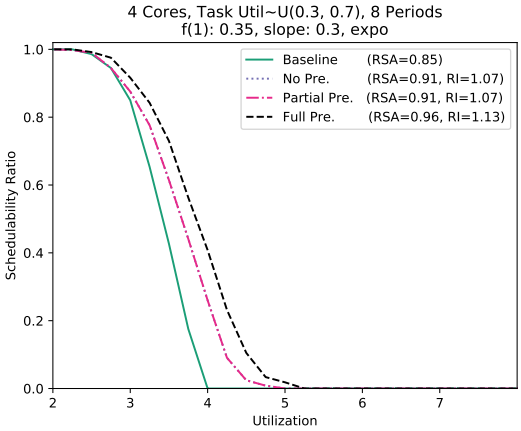

Figure 5.32: A median scenario by RI for partial and no preemption.

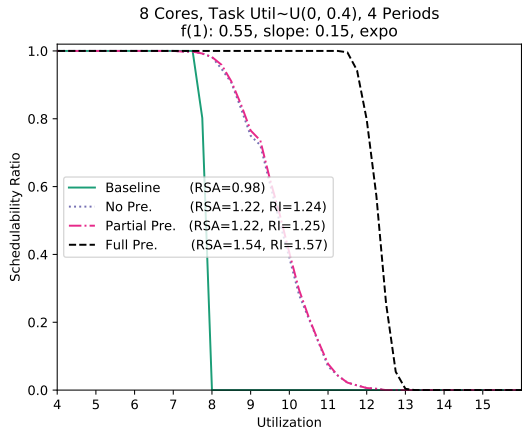

Figure 5.33: Light per-task utilization and average SMT parameters.

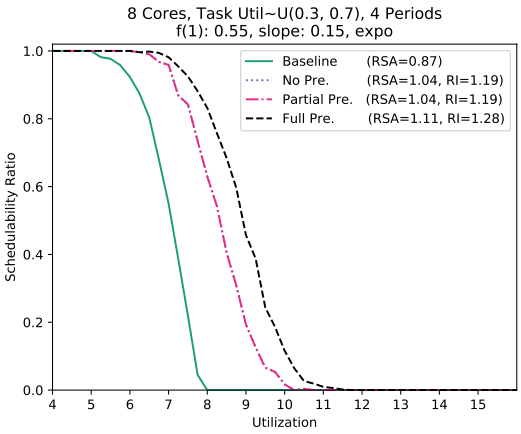

Figure 5.34: Medium per-task utilization and average SMT parameters.

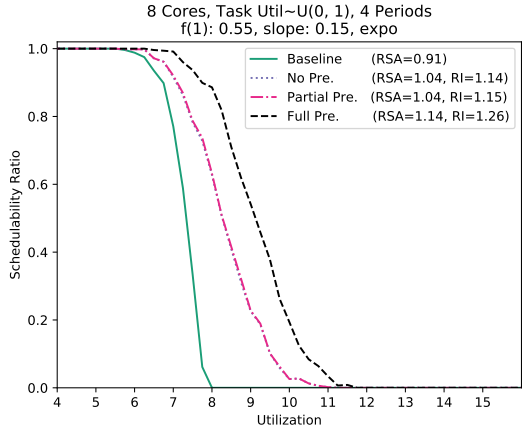

Figure 5.35: Wide per-task utilization and average SMT parameters.

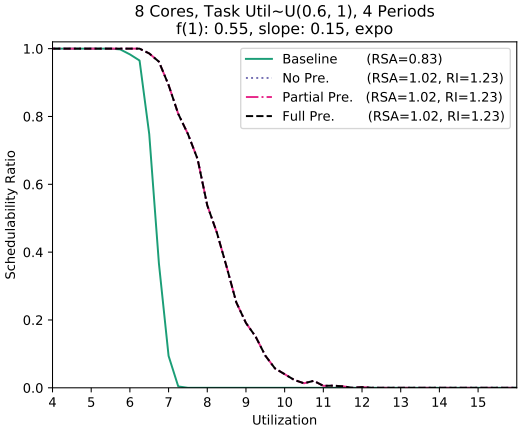

Figure 5.36: Heavy per-task utilization and average SMT parameters.

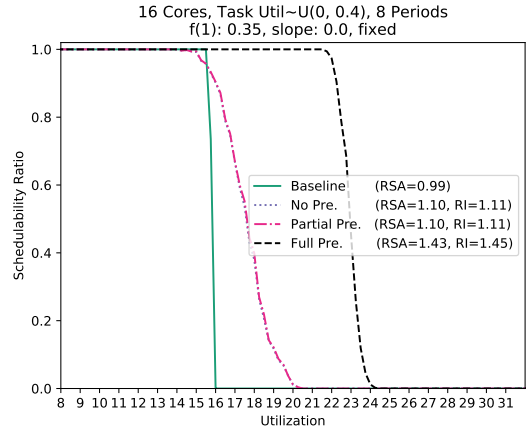

Figure 5.37: Identical parameters to Graph 5.28 apart from core count.

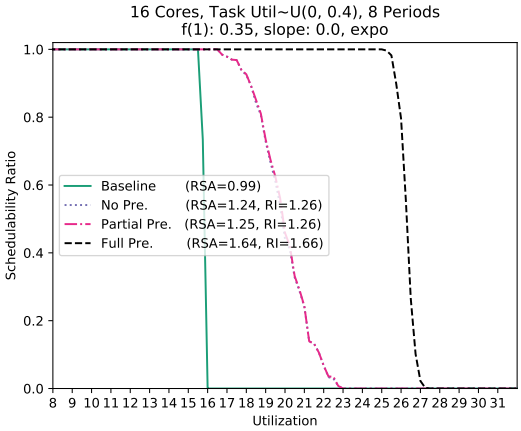

Figure 5.38: Identical to Graph 5.37 apart from using the exponential distribution.

Observation 5.12. *SMT is beneficial even given pessimistic assumptions about its timing effects.* When using our most pessimistic SMT model—$\mu = 0.75$ and slope of 0.3—SMT was beneficial in all cases for full preemption, as shown in Figures 5.29 and 5.30. For the no preemption and partial preemption models, the median RI given those SMT parameters was 1.02. Figures 5.29 and 5.30 show RI values for no preemption and partial preemption of 1.01 and 1.03, respectively.

Observation 5.13. *There is little difference between limited and no preemption.* In all of our graphs shown, the lines for limited and no preemption scheduling are either identical or very close. In the graphs that appear to have only three schedulability curves, the two are surimposed. Limited preemption would be significantly different from no preemption only if $C_{i(j)}^{(1)}$ and $C_{j(i)}^{(1)}$ are dissimilar for many paired tasks $\tau_{i:j}$. However, since utilization is reduced the most when tasks have similar costs, our optimization program will tend to pair tasks with similar execution times.

Observation 5.14. *SMT becomes more effective as core counts increase.* For example, Figure 5.37 differs from Figure 5.28 only by the core count, but shows vastly improved SMT effectiveness. For the full-preemption model, two effects are in play: more tasks means better opportunities for the optimization program to find good pairs, and partitioning is more effective on larger platforms, even without SMT. For the partial and no preemption models, an additional difference is that in Figure 5.37, cores outnumber periods.

Observation 5.15. As with CERT-MT, *SMT is more beneficial when using the exponential than the fixed distribution.* Our best graph, Figure 5.27, uses the exponential distribution, whereas all three of our worst graphs, 5.28, 5.29, and 5.30, used the fixed distribution. In addition, compare Figures 5.37 and 5.38, which differ only in that respect; Figure 5.38 is clearly better.

Observation 5.16. For all three preemption models, *changing the slope of f_i generally had a smaller impact than changing the expected value of $f_i(1)$.* For example, see Figures 5.39 through 5.42. When comparing Figure 5.39 to Figure 5.41, RSAs for all three preemption models drop by 0.07 or more as $f_i(1)$ increases from 0.35 to 0.55. When comparing Figure 5.40 to 5.42, RSA drops by 0.1 for the full preemption model and by 0.06 for the other two models.

The decrease in RSA is much less when comparing graphs with the same $f(1)$ values but different slopes, i.e., comparing graphs on the left and right. Between Figures 5.39 and 5.40, no RSA decreases by more than 0.6; the same is true between Figures 5.41 and 5.42. A similar result was seen for CERT-MT in Section 5.2.

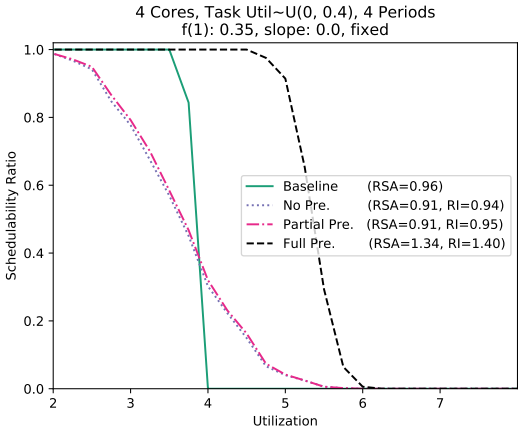

Figure 5.39: $E[f(1)] = 0.35$ with slope 0.0.

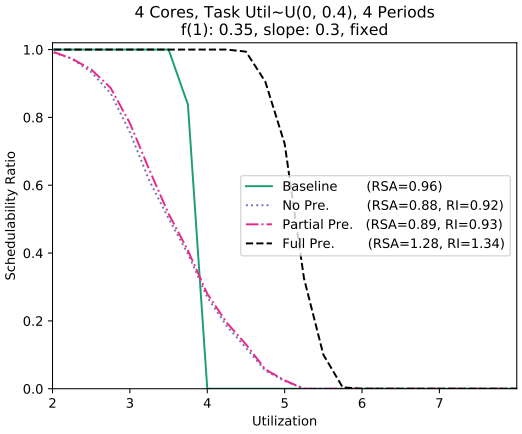

Figure 5.40: $E[f(1)] = 0.35$ with slope 0.3.

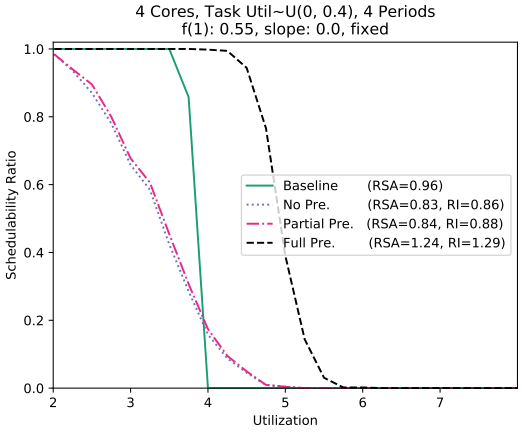

Figure 5.41: $E[f(1)] = 0.55$ with slope 0.0.

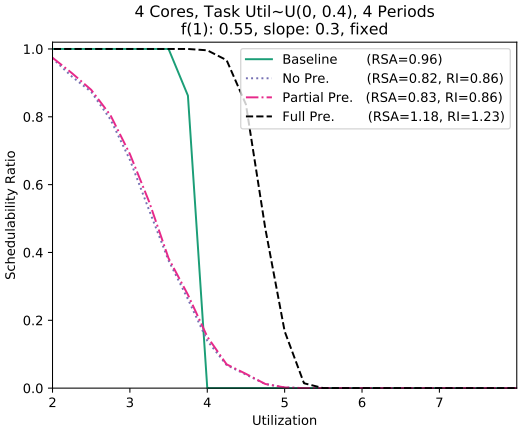

Figure 5.42: $E[f(1)] = 0.55$ with slope 0.3.

Observation 5.17. *In no case did our preemption program require more than three seconds to execute.* For every scenario, we recorded the maximum time for the preemption program to run on any task system in that scenario; all values were under one second. We executed sets of twelve scenarios in parallel on 24-core nodes of a large research cluster. We did not dictate how multiple cores were were to be used. However, even if 24 cores were used in parallel, we would still expect that optimization programs such as those used here could be run on less than ninety seconds on a single core.

As with Section 5.2, scenarios were tested on a shared cluster. For this reason, times needed to schedule scenarios may vary considerably from our observations.

5.5 Conclusions and future work

In this chapter, we have given two algorithms for using SMT to support HRT systems while using simultaneous co-scheduling. We have shown, within the context of our schedulabilty study, that SMT can, in the best cases, allow a given platform to schedule systems with utilization more than 1.5 times greater than what it could schedule without SMT.

In future work, we plan to look for algorithmic shortcuts to make the CERT-MT scheduler more practical and larger systems and improve the ability of our priority-driven scheduler to manage non-preemptive task systems.

CHAPTER 6: APPLYING SMT TO HRT DAGS

So far, we have only considered systems where all tasks are strictly sequential, i.e., no job may execute in parallel with itself. In this chapter, we show how to use SMT to support HRT DAG tasks, which were described in Section 2.4.1. In Section 6.1 we give a method for using SMT to minimize the utilization of a single DAG by simultaneously co-scheduling selected subtasks. In Section 6.2, we evaluate our method via a schedulability study. We consider both the effects of SMT on individual DAGs and the effects of SMT on multi-DAG systems. In Section 6.3, we conclude and give directions for future work. This chapter is based on work first done in 2022 (Osborne et al., 2022).

6.1 Applying SMT to DAG Tasks

SMT can be used to improve the schedulability of DAG tasks by executing vertices in parallel on a single core. Informally, our approach can be described as using SMT to make DAGs skinnier. Figures 6.1 and 6.2 (duplicates of Figures 1.2 and 1.3) show a DAG before and after using SMT to combine vertices v_2 and v_5.

To decide which vertices to combine, we use an optimization program that minimizes total utilization—we will shortly redefine utilization to account for SMT—without violating precedence constraints or making the DAG infeasible. A DAG is infeasible if its deadline is less than its length, i.e., $D < L$ holds. After applying this program, we use Algorithm 1 to determine how many cores are required to schedule the DAG, with combined vertices being simultaneously co-scheduled on a single core. Essentially, we are reducing utilization as a heuristic to reduce the number of cores required.

6.1.1 Cost and Utilization for DAGs using SMT

To denote costs, we define $c_{i(j)}$ and $c^{+}_{i(j)}$ identically to $C^{(1)}_{i(j)}$ and $C^{+}_{i(j)}$ (Definitions. 3.9 and 5.3); the only difference is that here we are considering subtasks rather than individual tasks.

Definition 6.1. Let c_i be the cost to execute subtask v_i. Unless stated otherwise, c_i is v_i's WCET. ◄

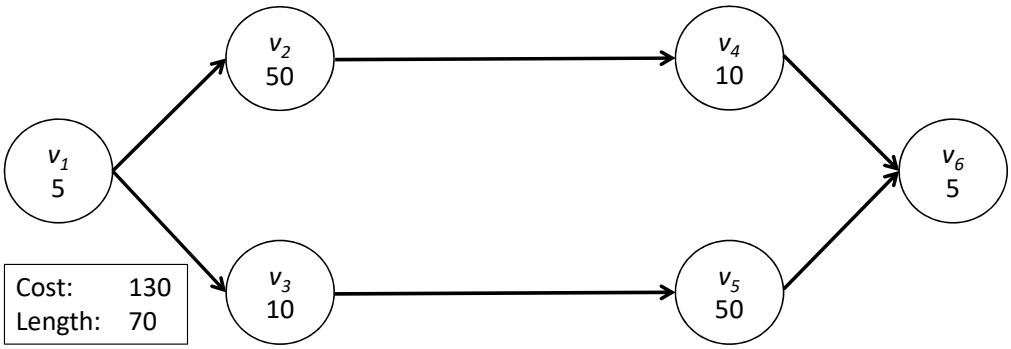

Figure 6.1: A DAG task consisting of six subtasks.

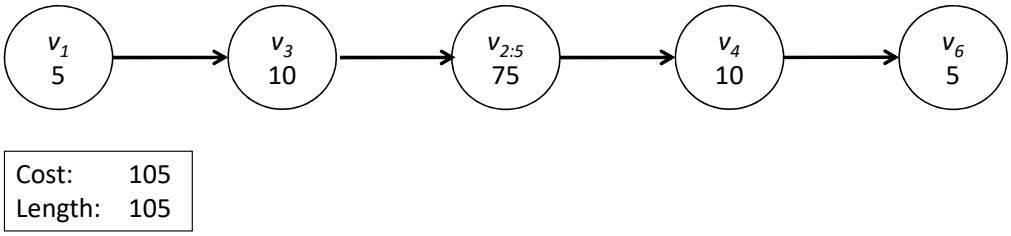

Figure 6.2: The same DAG task; subtasks v_2 and v_5 paired via SMT.

Definition 6.2. Given subtasks v_i and v_j, let $c_{i(j)}$ be the cost to execute v_i given it is simultaneously co-scheduled with v_j, $c_{j(i)}$ the cost to execute v_j given it is simultaneously co-scheduled with v_i, and $c^+_{i(j)}$ the maximum of $c_{i(j)}$ and $c_{i(j)}$. If $i = j$, then all three values are equal to c_i. ◀

Because we are considering DAG scheduling only with simultaneous co-scheduling, we do not include the (1) superscript on $c_{i(j)}$. We use $c^+_{i(j)}$ to give an SMT-aware definition of a DAG's utilization.

Definition 6.3. The utilization of a DAG task is given by

$$U = \sum_{\forall i: v_i \text{ is a solo task}} \frac{c_i}{T_i} + \sum_{\forall i,j: i>j, v_i \text{ and } v_j \text{ are paired}} \frac{c^+_{i(j)}}{T_i}. \blacktriangleleft \tag{6.1}$$

6.1.2 Optimization Decision Variables and Objective

To mathematically denote which vertices should be paired, we use a binary decision variable.

Definition 6.4. Let $x_{i:j}$ be defined to equal 1 if subtasks v_i and v_j are paired and 0 otherwise. For $i = j$, let $x_{i:j}$ equal 1 if v_i is to be executed without SMT and 0 otherwise. In all cases, we require that $x_{i:j} = x_{j:i}$ holds. We refer to these variables as the x variables. ◀

Combining Definitions 6.3 and 6.4 allows us to express a DAG's utilization in terms of the x variables. Recall from Section 2.4.1 that V is the set of all vertices in a DAG and each vertex corresponds to a subtask; $|V|$ is therefore the subtask count.

$$U = \sum_{i=1}^{|V|} \sum_{j=i}^{|V|} x_{i:j} \cdot \frac{c^{+}_{i(j)}}{T} \tag{6.2}$$

We use a second set of decision variables to track possible start and finish times for each subtask in the presence of SMT.

Definition 6.5. Let s_i and f_i give possible start and finish times, respectively, for subtask v_i. ◀

Note that s_i and f_i are only *possible* start and finish times and may not correspond to how v_i is eventually scheduled. Their purpose is not to construct a schedule, but to ensure that pairing subtasks does not create a DAG that is infeasible.

We define our optimization program as minimizing Expression (6.2) subject to the constraints of Section 6.1.3.

6.1.3 Optimization Constraints

(i) **Every subtask executes either alone or with at most one other subtask.** This restriction guarantees that all subtasks actually execute and implements the requirement of simultaneous co-scheduling (Definition 3.8) that every task can be co-scheduled with at most one other task.

$$\forall i : \sum_{j=i}^{|V|} x_{i:j} = 1$$

(ii) **Paired subtasks begin simultaneously.** This restriction is required by Definition 3.8. For unpaired v_i and v_j, where $x_{i:j} = 0$, the expression below is trivially true.

$$\forall i, j : s_i \cdot x_{i:j} = s_j \cdot x_{i:j}$$

(iii) **All subtasks finish only after executing for sufficient time.** If v_i and v_j are paired, then v_i finishes at time $f_i = s_i + c_{i(j)}$. Notice that f_i is only dependent on s_i and $c_{i(j)}$, not on $c_{j(i)}$.

$$\forall i : f_i = s_i + x_{i:j} \cdot c_{i(j)}$$

(iv) **All subtasks finish prior to the DAG's deadline D**. This restriction preserves feasibility.

$$\forall i : f_i \leq D$$

(v) **Precedence constraints are respected.** No subtask begins until all of its predecessors have completed.

$$\forall i : \forall j :: v_j \text{ precedes } v_i : s_i \geq f_j$$

Rule (v) does not require that a job waits until any subtasks paired with its predecessors have completed.

Example 6.1. Suppose that v_1 and v_2 begin executing at time 0 and are paired together; $c_{1(2)} = 10$, and $c_{2(1)} = 15$. Let v_3 have v_1 but not v_2 as a predecessor. Assuming a core is available at time 10 when v_1 completes, v_3 may begin executing at time 10 and does not need to wait for v_2 to finish at time 15. $\Diamond$

Notice that (v) combined with (ii) prohibits precedence-constrained tasks from executing as pairs.

The optimization program outputs pairing decisions via the value of all x variables. Pairs can then be assigned to cores using Algorithm 1 or another DAG-scheduling algorithm. While this optimization program is not optimal with respect to minimizing core count, we have found that for large-utilization DAGs, reductions in core count closely track reductions in utilization made using our methods. We further discuss this in Section 6.2.

Limiting the number of variables. The optimization program's complexity is exponential relative to the number of variables, and the number of x variables needed is proportionate to $|V|^2$. Thus for large DAGs optimization may be unmanageable. For those cases, we provide an additional restriction.

(vi) **Only subtasks with indices that differ by at most some constant K can be paired.**

$$|i - j| > K \rightarrow x_{i:j} = 0$$

In practice, not defining $x_{i:j}$ when $|i-j| > K$ provides a greater performance benefit than restricting $x_{i:j}$ to zero. Note that K can be at most $|V|$.

In our experiments, $K = 10$ allowed our optimization program to handle DAGs with as many as one hundred subtasks while still giving near-optimal reductions in utilization. The effects of different K values are discussed more in Section 6.2.2.

6.2 Applying SMT to HRT DAGs: Schedulability Studies

In this section, we present our synthetic task experiments and results. We simulated over 70 thousand DAGs and over four million subtasks across more than one thousand scenarios. For each scenario, we test the ability of our optimization program to reduce per-DAG utilization and, when combined with Algorithm 1, to reduce the cores required per DAG. We consider individual DAGs first, followed by systems of DAGs using federated scheduling.

6.2.1 Experimental Setup for Single-DAG Systems

Each scenario is defined by the number of subtasks $|V|$ per DAG, possible costs per subtask, a model for SMT behavior, a model for precedence constraints, and K. Scenario parameters are explained in more detail below and summarized in Table 6.1.

Subtask count. For each scenario, we selected $|V|$ from $\{10, 20, 40, 80, 100\}$.

Baseline costs. To determine the costs of individual subtasks without SMT, each scenario had costs selected from a *narrow* $(1-2)$ or *wide* $(1-20)$ uniform distribution. Our expectation was that SMT would provide a greater benefit with the narrow range, as our model prohibits pairs where c_i and c_j differ by an order of magnitude, and pairing tasks with very different costs provides less overall utilization benefit. This expectation was confirmed.

SMT behavior. Our model for SMT is identical to that of Section 3.2 and Chapter 5. For all scenarios, we considered all three possible values—0.35, 0.55, and 0.75—of $E[f(1)]$. For scenarios with $|V| \leq 20$, we also considered all three possible slope values, 0.0, 0.15, and 0.3. For $|V| > 20$, we used a slope of 0.15 in all scenarios.

Precedence constraints. After assigning costs to all subtasks within our DAG, we use one of two methods to determine precedence constraints within our DAG.

In the *Erdős-Rényi* method, every pair of subtasks have a probability p of being connected by an edge (Erdős and Rényi, 1960). Lower values of p will produce DAGs with more potential for parallelism. We use the p values $\{0.1, 0.3, 0.5\}$. In preliminary experiments, we found that DAGs with $p > 0.5$ had very few subtasks executable in parallel and thus received little to no benefit from SMT. We did not use this method with 80- or 100-subtask DAGs.

In the *layer-by-layer* method, each DAG is first divided into ℓ layers. No precedence constraints ever exist between subtasks within the same layer, but subtasks in different layers have probability p of being connected by an edge (Tobita and Kasahara, 2002). We use the p values $\{0.5, 0.75, 1\}$. While $p = 1$ specifies a task with no parallelism in the Erdős-Rényi method, in this case it means that all subtasks in one layer must complete before the next layer can begin.

For ℓ, we use the values $\{2, 4, 8, 16\}$, with the caveat that $\frac{|V|}{\ell} \geq 5$ must hold, i.e., the expected number of subtasks per layer must be at least five. To divide the subtasks into layers, we randomly select $\ell - 1$ integers from the range $[1, |V|)$ without replacement. These integers subdivide the set of tasks into layers by index; each selected integer gives the index of the last task in a layer. $|V|$ is excluded from the selection range; if it were selected, the DAG would have one less layer than intended.

Example 6.2. Suppose we wish to build a DAG of 20 subtasks in three layers. Two values are chosen from the range $[1, 20)$. If we select 3 and 10, the first layer consists of subtasks v_1 through v_3, the second of subtasks v_4 through v_{10}, and the third layer of all remaining subtasks. ◊

This method was designed to make the DAG's total length L easily controllable; no chain in a DAG created this way will have more than ℓ members (Tobita and Kasahara, 2002). Setting $\ell = |V|$ is equivalent to using the Erdős-Rényi method. Further information regarding the properties of both methods, as well as additional DAG-generation methods, can be found in (Cordeiro et al., 2010).

DAG utilization and deadline. Each DAG's total utilization is selected uniformly from the range $\left[1, \frac{C}{L}\right]$. After doing so, we give each DAG a deadline of $D = \frac{C}{U}$ (we continue to assume implicit deadlines, i.e., $D = T$). We do not consider DAGs with $U < 1$ as those can be scheduled sequentially on a single core without parallelism. DAGs for which $U > \frac{C}{L}$ (equivalently, $D < L$) holds are not schedulable on any number of cores, with or without SMT. Note that $\frac{C}{L}$ is itself a function of individual DAG structures. Consequently, the distribution of utilization across all DAGs in a scenario will not be uniform.

Tunable parameter K. Our final parameter is the value of K as used in restriction (vi) of the optimization program. We use K values from the set $\{1, 10, 20, 40\}$ except in the 80- and 100-subtask scenarios, where only $K \in \{1, 10\}$ was used. For 40-subtask scenarios, we used all K values with the layer-by-layer method, by only $K \in \{1, 10\}$ for the Erdős-Rényi scenarios. We did not allow K to exceed the subtask count; doing so would be meaningless. The optimization program is only optimal when restriction (vi) is removed by setting $K \geq |V|$, but we observed excellent results for all $K \geq 10$.

Solution limit. Unlike our experiments of Section 5.2, we did not impose a time limit to find a solution. However, we did limit Gurobi to finding 10 solutions per DAG; if none of the first solutions found could be guaranteed to minimize utilization, then we used the best solution of the first 10.

In total, we had 432 10-subtask DAGs, 972 20-subtask DAGs, 504 40-subtask DAGs, and 288 DAGs each for the 80- and 100-subtask scenarios.

6.2.2 Single DAG Results

We summarize the optimization program's effect on each scenario using three metrics: *relative core count*, *relative utilization*, and *core reduction frequency.*

Definition 6.6. *Relative core count* (respectively, *relative utilization*) is defined as a DAG's required core count, per Algorithm 1 (respectively, total utilization) with SMT divided by the same values without SMT. They are abbreviated as RCC and RU. ◄

Values of 1.0 indicate no change due to SMT; values less than 1.0 indicate SMT has reduced the DAG's utilization or core count requirement.

Definition 6.7. *Core reduction frequency* is defined as the number of cases where the required core count was reduced by at least one divided by the total number of cases evaluated. It is abbreviated CRF. ◄

While smaller values are better for RCC and RU, larger values are better for CRF.

In rare cases, we found that applying SMT would *increase* the number of cores needed. In these cases, the best choice is to not use SMT, so we set RU and RCC to one and CRF to zero.

We summarize each scenario with a scatterplot that plots SMT utilization, baseline cores, and SMT cores on the vertical axis against baseline utilization on the horizontal axis. Each individual DAG is shown by three vertically aligned symbols: the purple x shows utilization after applying SMT, the green diamond shows the number of cores required after applying SMT, and the orange circle shows the number of cores required

without SMT. The position of the three symbols along the horizontal axis indicates the DAG's total utilization without using SMT.

In the majority of scenarios, SMT utilization appears to be a linear function of baseline utilization; in these cases, RU approximates the function's slope. For reference, we include in each graph a solid black line with slope 1.0. Reporting our results using schedulability curves and RSA values, as in Chapters 4 and 5, was not practical here due to the large number of experiments per scenario that are required to produce high-quality schedulability curves.

Example 6.3. Figure 6.6 summarizes the effects of SMT on DAGs with 10 subtasks with $\ell = 2$ and $p = 1$ when using $K = 10$ (first line of graph title). SMT is modeled by setting $f(1) = 0.35$, using a slope of 0.15, and the exponential distribution (second line of title). On average, applying SMT reduces DAG utilization to 74% of what it had been without SMT and reduces core requirements to 73% of what they would have been without SMT (RU and RCC values, third line of title). 65% of DAGs tested had some reduction in the required number of cores (CRF, third line of title).

Again, each DAG considered is represented by three vertically aligned symbols. Consider the three rightmost symbols in the graph. They indicate a DAG with $U \approx 6.75$, indicated by the symbols' position on the horizontal axis. Without SMT, scheduling this DAG required ten cores, as indicated by the orange dot in the graph's top right corner. With SMT, this graph required seven cores, as indicated by the height of the rightmost green diamond. Finally, the DAG's utilization after applying SMT is approximately 4.5 and is indicated by the position of the rightmost purple x.

Looking at the graph as a whole, RU is approximately equal to the slope of a linear function approximating utilization with SMT—purple x symbols—as a function of baseline utilization. The black line has a slope of 1.0, making it easier to see this relationship visually.

RCC corresponds to the average vertical distance between orange dots, indicating cores required without SMT, and green diamonds, indicating cores required with SMT. Finally, CRF is equal to the proportion of orange dots that are above, rather than superimposed on, a green diamond. ◊

Execution times. We tracked solver runtimes for each scenario. As in Sections 5.2 and 5.4, we executed the optimization program on a shared cluster, so our recorded runtimes should be considered mainly for order-of-magnitude comparisons. Recorded times are for serial execution; we used parallelism to analyze many DAGs at once, but not to reduce the analysis time for individual DAGs. Our per-DAG runtimes ranged

Table 6.1: Summary of Parameters used in single-DAG schedulability tests

Parameter Description	Parameter Symbol	Possible Values
expected $M_{i(j)}^{(1)}$ given $C_i \leq C_j$	$E[f(1)]$	0.35, 0.55, 0.75
slope of regression function	none	0.0, 0.15, 0.30
distribution for $M_{i(j)}^{(1)}$ given f_i	none	fixed, exponential
per-subtask cost distribution	none	$U(1,2)$ (narrow) $U(1,20)$ (wide)
subtasks per DAG	none	10, 20, 40, 80, 100
max index difference between paired subtasks	K	1, 10, 20, 40
number of layers	ℓ	2, 4, 8, 16 Erdős-Rényi
Probability of edge between subtasks	p	0.5, 0.75, 1.0 (with layers) 0.1, 0.3, 0.6 (with Erdős-Rényi)

from less than a second to over four days. The total CPU time for analyzing all DAGs was roughly four years. Like our other experiments, these experiments were conducted on a shared research cluster; observed execution time requirements are useful mainly for order of magnitude comparisons.

Our full set of graphs is included in Appendix D. Here, we show graphs that are representative of our best and worst results or that demonstrate particularly interesting behaviors. Each graph's title identifies its scenario; if the number of layers is unstated, then the graph was created using the Erdős-Rényi method. In addition, graph titles summarize performance by stating RU, RCC, CRF, and mean runtime in seconds.

6.2.2.1 DAGs With at Most 80 Subtasks

We first focus on scenarios where $|V| \leq 80$ holds. Scenarios with 100 subtasks are considered separately.

We classify scenarios as either *good*, *moderate*, or *poor* based on mean RCC. $RCC < 0.8$ is good; $RCC > 0.9$ is poor; and intermediate RCC values are moderate. Among the 2,196 scenarios with at most 80 subtasks per DAG, we had 208 good scenarios (9%), 954 moderate scenarios (48%) and 769 poor scenarios

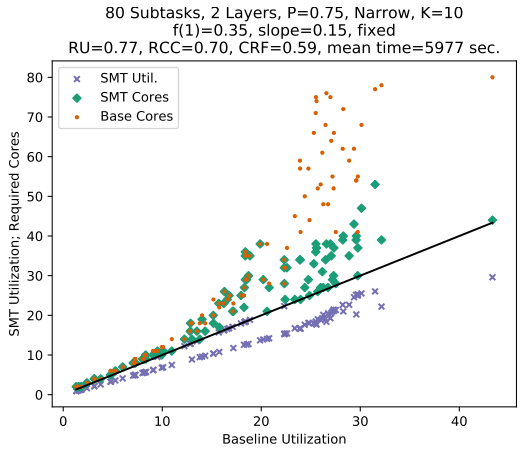

Figure 6.3: Best 80 subtask scenario.

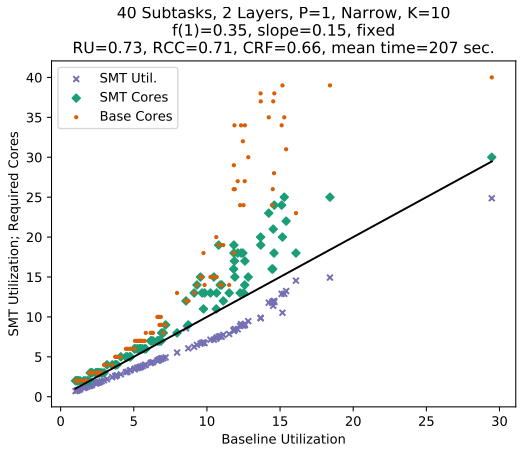

Figure 6.4: Best 40 subtask scenario.

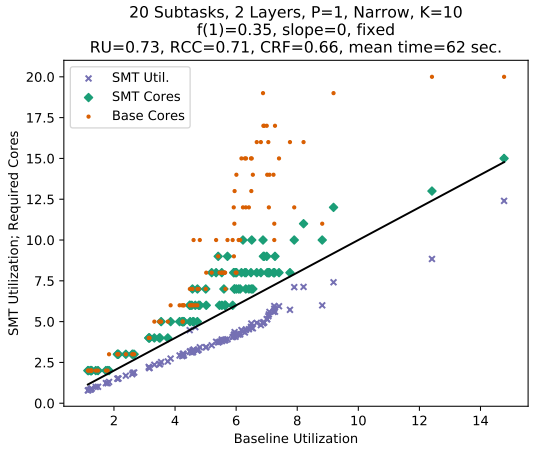

Figure 6.5: Best 20 subtask scenario.

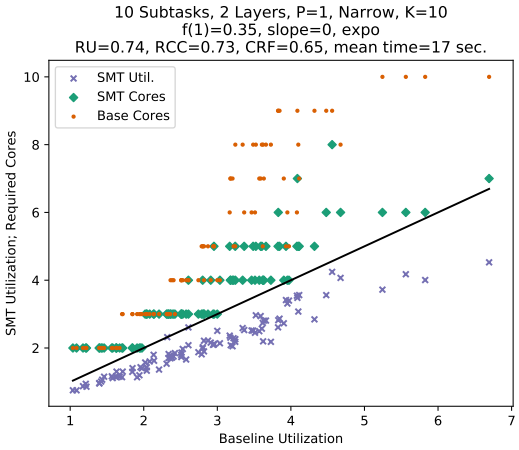

Figure 6.6: Best 10 subtask scenario.

(43%). When we excluded scenarios for which $K = 1$ held, 1368 scenarios remained. Of those, 177 (13%) were good, 707 (52%) were moderate, and 484 (35%) were poor.

Observation 6.1. *In the best scenarios, core count was reduced, on average, to approximately 70% of what would be required without SMT.* Examples include Figures 6.3 through 6.6. In the best cases for individual DAGs, core count was reduced by as much as a third, seen in the right sides of Figures 6.3 through 6.6 and Figures 6.17 through 6.20.

Observation 6.2. *It is possible for core count to decrease significantly more than total utilization.* Examples can be seen in Figures 6.7 and 6.8. This case occurs when small reductions in utilization enable a DAG to be scheduled on fewer cores. For example, a DAG with $U = 1.01$ requires at least two cores under federated

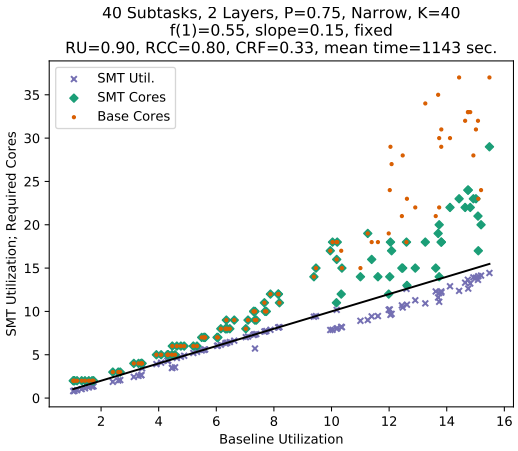

Figure 6.7: Example of core count reduced more than utilization

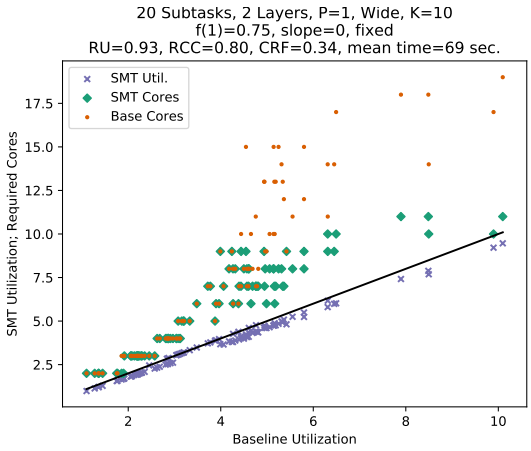

Figure 6.8: A second example of core count reduced more than utilization

scheduling, but reducing its utilization to 0.99 would allow it to execute on a single core. The opposite situation—a high RCC value despite a low RU value—can be seen in Figure 6.9.

Observation 6.3. *The greatest benefits to core count come from combining subtasks that, without SMT, would require dedicated cores to execute.* For example, the scenarios of Figures 6.3 through 6.6 and 6.17 through 6.20 all include cases where the baseline core requirement approaches the total number of subtasks. Conversely, in the scenarios of Figures 6.9 through 6.12, the number of cores required is much less than the subtask count even without SMT, giving less room for improvement.

Without SMT, two subtasks with combined $U > 1$ cannot be scheduled on a single core; with SMT, scheduling the two together becomes possible if the corresponding pair has $U \leq 1$. In a DAG that consisted entirely of subtasks with $U = 0.51$, SMT could plausibly halve the required core count.

Observation 6.4. *Core count was reduced more when DAG utilization was large.* This effect can be seen clearly in Figures 6.13 through 6.16, where all parameters apart from the number or presence of layers remain constant. Our good scenarios—Figures 6.3 through 6.6 and 6.17 through 6.19—generally have higher per-DAG utilizations than our poor scenarios (Figures 6.9 through 6.12). Within each graph, even those for poor scenarios, core counts are reduced most on the right side of the graph, where utilization is largest.

There are two ways to interpret this result. First, when utilization is larger, the change needed to reduce core count is relatively small. Second, recall that we determine utilization by assigning each DAG a utilization in the range $\left[1, \frac{C}{L}\right]$, where C is the DAG's total cost and L its length. If C is much greater than L, then it is

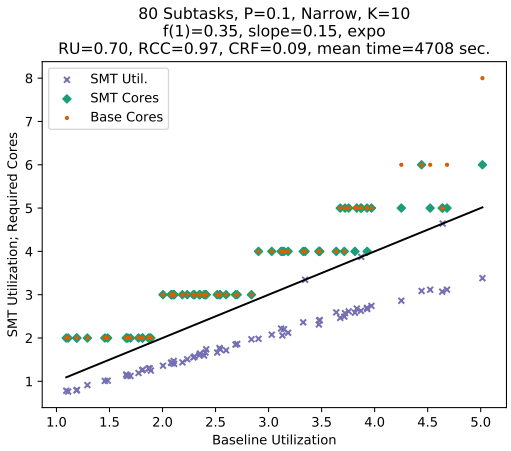

Figure 6.9: Poor RCC despite good RU.

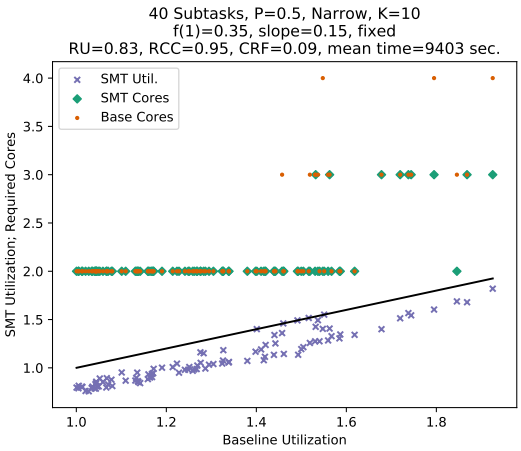

Figure 6.10: Erdős-Rényi with $p = 0.5$.

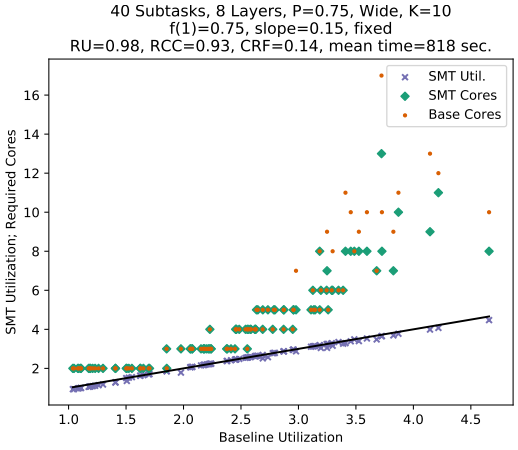

Figure 6.11: Poor performance using the layer-by-layer method.

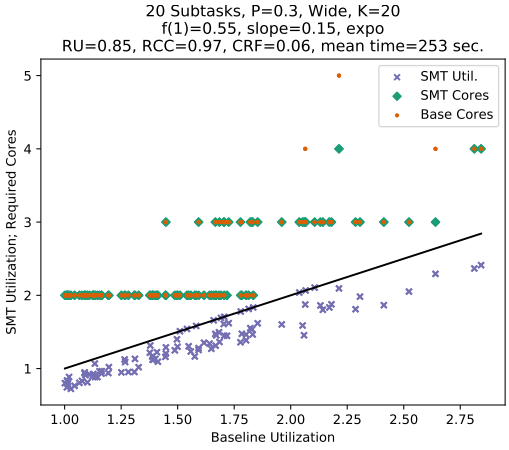

Figure 6.12: Erdős-Rényi with $p = 0.3$.

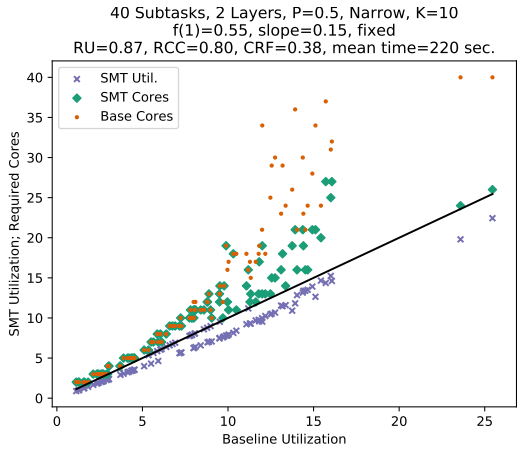

Figure 6.13: Layer-by-layer: 2 layers.

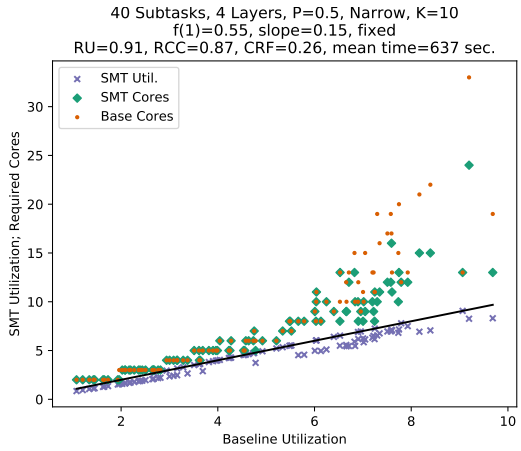

Figure 6.14: Layer-by-layer: 4 layers.

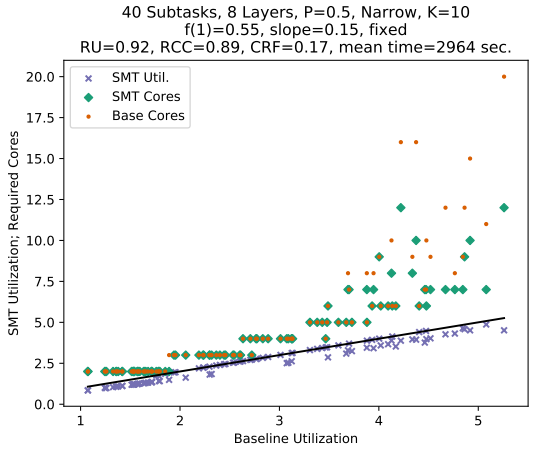

Figure 6.15: Layer-by-layer: 8 layers.

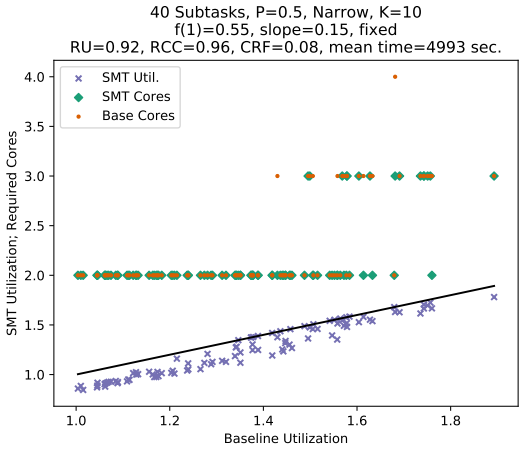

Figure 6.16: Erdős-Rényi method.

possible for the DAG to have greater utilization and the DAG's structure includes more potential parallelism that can be exploited by SMT.

Observation 6.5. *The scenario parameters yielding the smallest RCC values were those that produced highly parallel DAGs.* This observation is closely related to Observation 6.4. All two-layer DAGs fell into either the good or moderate RCC range, as did almost all DAGs created using the Erdős-Rényi method with $p = 0.1$. Figures 6.13 through 6.16 illustrate this effect by changing the number of layers while keeping all other parameters constant.

Observation 6.6. *RCC values were generally better for larger subtask counts.* For 10, 20, 40, and 80 cores the minimum RCC values, respectively, were 0.725, 0.706, 0.710, and 0.697; scenarios with these values are

shown in Figures 6.3 through 6.6. This observation is generally due to the layer-by-layer method; when using the layer-by-layer method, higher subtasks counts allow more tasks per layer, and thus more parallelism.

Observation 6.7. *Decreasing K from $|V|$ to as low as 10 allowed SMT to remain beneficial while dramatically reducing runtime.* Setting $K = 1$ is more detrimental, but is better than not using SMT at all. Figures 6.17 through 6.20 give examples. With $K = 40$ (Figure 6.17) per-DAG execution time requires an average of 716 seconds. Reducing K to 20, 10, and 1 reduces execution time to 689 seconds, 189 seconds, and 102 seconds, respectively. Reducing K from 40 to 20 leaves RCC unchanged at 0.80. Setting $K = 10$ causes RCC to decrease to 0.76, but setting $K = 1$ causes RCC to increase to 0.84.

We were surprised to see that both RCC and RU are the smallest when $K = 10$. In theory, our optimization program is optimal only when $K = |V|$ holds. However, recall that to keep runtimes manageable, we limit the solutions per DAG to 10. We suspect that optimal solutions involve pairing together subtasks with similar indices and that having a relatively small K value helps the solver avoid sub-optimal solutions.

Observation 6.8. *Using the exponential distribution to determine $M_{i(j)}$ did* not *result in significantly better performance than using the fixed distribution.* In fact, we see some weak evidence that DAGs built using the fixed distribution saw lower RCCs than those using the exponential distribution. This is the opposite of what we observed in other HRT cases. In Chapter 4, the best scenarios were invariably those that used the exponential distribution. Here, the fixed distribution leads to lower RCC values in the best case for the cases of 80, 40, and 20 subtasks (Figures 6.3, 6.4, and 6.5).

Observation 6.9. *Scenarios using the wide per-subtask cost distribution generally out-performed those using the narrow distribution.* This result is not surprising, as pairing subtasks will tend to be more effective when they have similar costs. For example, notice that all four of our best scenarios in Figures 6.3 through 6.6 use the narrow distribution.

6.2.2.2 DAGs with 100 Subtasks

We tested 288 scenarios with 100-subtask DAGs; 144 each with $K = 1$ and $K = 10$. Unfortunately, we found that when $K = 10$ held, many DAGs required hours or even days to analyze; per-DAG execution times ranged from 20 seconds to more than five days. Our $K = 10$ results were weaker than those for smaller DAGs, but we still saw some improvement due to SMT; we had 14 good scenarios, 50 moderate scenarios, and 80

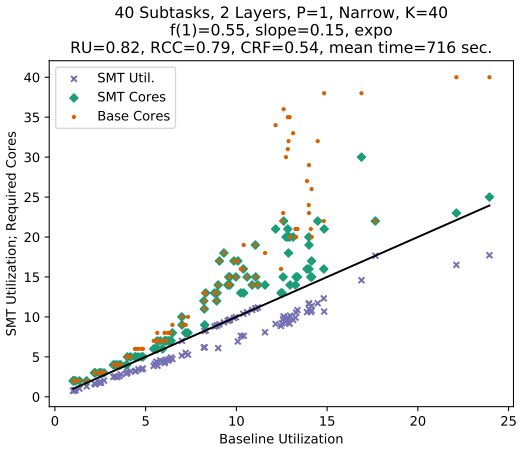

Figure 6.17: $K = 40$.

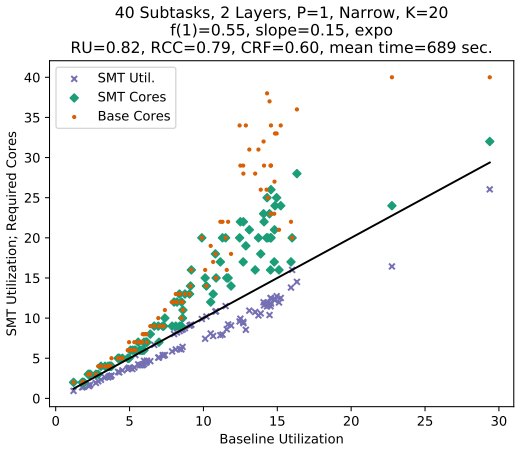

Figure 6.18: $K = 20$.

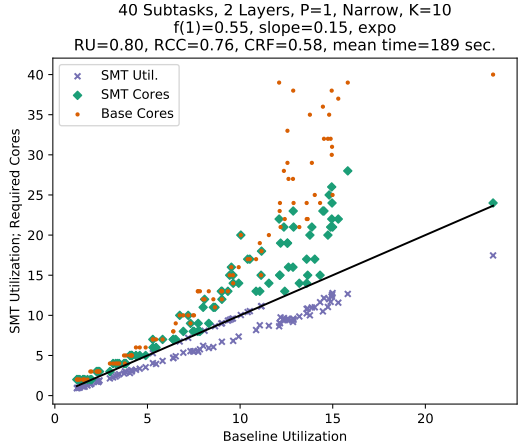

Figure 6.19: $K = 10$.

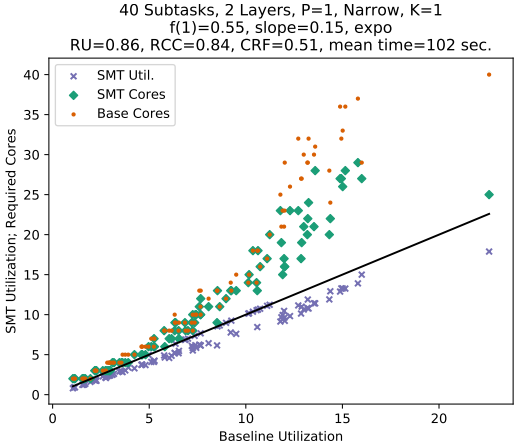

Figure 6.20: $K = 1$.

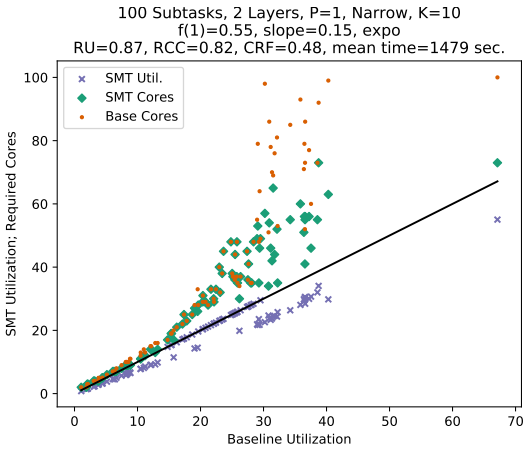

Figure 6.21: $|V| = 100$, 2 layers, $p = 1.0$, $K = 10$.

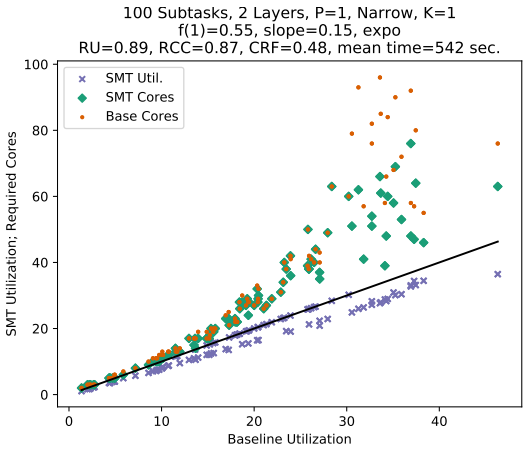

Figure 6.22: $|V| = 100$, 2 layers, $p = 1.0$, $K = 1$.

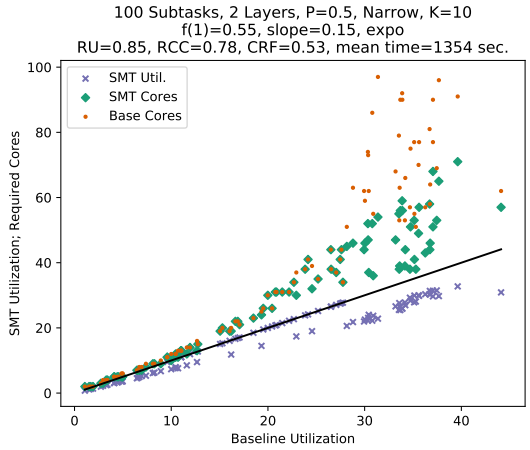

Figure 6.23: $|V| = 100$, 2 layers, $p = 0.5$, $K = 10$.

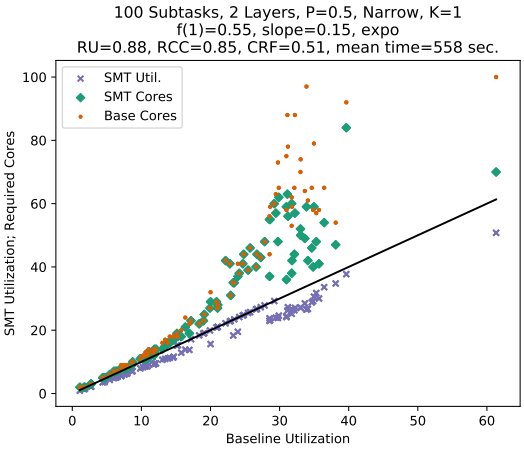

Figure 6.24: $|V| = 100$, 2 layers, $p = 0.5$, $K = 1$.

poor scenarios. For $K = 1$, execution times ranged from 5 seconds to 30 hours, but results were much poorer; we had only 9 good results, combined with 38 moderate and 97 poor results.

Our current algorithm was not designed for large DAGs. A better approach to using SMT for DAGs of this size or larger would be to first heuristically divide a DAG into sub-graphs of fewer than 100 vertices each and then apply our algorithm to the resulting sub-graphs. For DAGs that use the layer-by-layer approach, we suspect analyzing a DAG one layer at a time would be highly effective.

Observation 6.10. *When $K = 10$ held, results were slightly weaker than those for similar parameters for smaller $|V|$ and larger K values.* For example, Figure 6.21 has, apart from $|V|$, the same parameters as Figure 6.19. Figure 6.21 has an RCC 0f 0.82 and Figure 6.19 an RCC of 0.76.

Observation 6.11. *When $K = 1$ held, results were weaker than those for $K = 10$.* For example, compare Figure 6.22 (RCC 0.87) to Figure 6.21 (RCC 0.82).

6.2.3 Multi-DAG Systems and Federated Scheduling

In this subsection, we consider scheduling systems of multiple DAGs using federated scheduling, which we described in Section 2.4.1.

Creating systems of DAGs. Within each system, all DAGs are created using a single set of parameters. We used a subset of the parameters that were already discussed in Section 6.2.1 and summarized in Table 6.1. We considered DAGs consisting of 10, 20, or 40 subtasks each. For the 10- and 20-subtask DAGs, we used both the Erdős-Rényi method with $p \in \{0.1, 0.3, 0.5\}$ and the layer-by-layer method with $\ell \in \{2, 4, 8\}$ with $p \in \{0.5, 0.75, 1.0\}$. For the 40-subtask DAGs, we excluded the Erdős-Rényi method due to high execution time requirements. In all cases, we maintained the requirement that $\frac{|V|}{\ell} \geq 5$ holds.

We used both the narrow and wide per-subtask cost ranges. We continued to use all three possible values for $E[f(1)]$, but only considered 0.15 for the slope. We used $K = 10$ in all cases.

Per-DAG utilizations. We augmented our existing parameters by specifying categories for each individual DAG's utilization: *light, low-heavy, medium-heavy,* and *high-heavy*. Total utilizations within each category are randomly chosen from the uniform ranges

$$\left(\frac{1}{2}, 1\right],$$
$$\left(1, \min\left(2, \frac{C}{L}\right)\right],$$
$$\left(\min\left(2, \frac{C}{L}\right), \min\left(4, \frac{C}{L}\right)\right], \text{ and}$$
$$\left(\min\left(4, \frac{C}{L}\right), \min\left(8, \frac{C}{L}\right)\right],$$

respectively. Each DAG is then assigned a deadline equal to its total cost without SMT divided by the assigned utilization.

Defining the bounds based on the minimum of $\frac{C}{L}$ and a constant rather than using a constant alone is necessary to guarantee that every DAG can be scheduled; scheduling a DAG is impossible if $D > L$ holds, i.e., $U > \frac{C}{L}$. Including $\frac{C}{L}$ in our utilization did produce some distortions in our results for the medium-heavy and high-heavy ranges; we discuss those effects as they come up in our observations below. Results for

low-heavy DAGs do not seem to be affected; the parameters governing the degree of parallelism in our DAGs will generally produce DAGs where $\frac{C}{L} > 2$ holds.

Light DAGs can all be scheduled sequentially on a single processor, even without SMT. Within the context of federated scheduling, SMT may allow more light DAGs to be scheduled per processor.

All other DAGs, since they have $U > 1$, are considered heavy per Definition 2.12 and would, without SMT, require at least two dedicated cores each. Our expectation is that SMT will convert many of the low-heavy DAGs into light DAGs with $U \leq 1$. Medium-heavy and high-heavy DAGs will generally remain heavy even after SMT has been applied, but may require fewer cores.

The number of DAGs per system ranged from 1 to $\frac{32}{E[U]}$, where $E[U]$ gives the expected value of a single DAG's baseline utilization; essentially we created systems with total utilization up to approximately 32. For each step from 1 to $\frac{32}{E[U]}$, we created 10 systems. A scenario consists of all systems built using a single set of per-DAG parameters.

6.2.4 Federated Scheduling Evaluation

As with individual DAGs, we evaluate each scenario on the basis of RCC, RU, and CRF. Scenario results are shown with scatterplots similar to those for single-DAG scenarios. This time, each point represents a system of DAGs; utilization and cores needed are totals for all DAGs in each system.

For each heavy DAG, the cores required are again determined by Algorithm 1. For light DAGs, we used partitioned scheduling, i.e., each DAG is assigned to a single core, but each core can contain multiple light DAGs up to total utilization 1.0. Tasks are assigned to cores using a bin-packing based approach. For each system, we attempted both decreasing-best-fit and decreasing-worst-fit and used the result that required the fewest cores.

We tested 288 10-subtask scenarios, 432 20-subtask scenarios, and 432 40 subtask scenarios, for a total of 1152 scenarios. When we use the same criteria as before—a good RCC value is less than 0.8, a poor RCC value is greater than 0.9, and a moderate value is in between—we have 351 good scenarios (30%), 447 moderate scenarios (39%) and 354 poor scenarios (31%).

We include the scatterplots for eight federated scheduling scenarios. Figures 6.25 through 6.28 depict some of our best scenarios. The only parameter to vary across these four scenarios is per-DAG utilization range. Figures 6.29 through 6.32 depict some of our worst scenarios, again varying only the per-DAG utilization parameter. Graphs for all federated scenarios are included in Appendix E.

Table 6.2: RCC values by per-DAG utilization range

	Good	Moderate	Poor
Light	123 (43%)	92 (47%)	13 (10%)
Low-heavy	106 (37%)	127 (44%)	55 (19%)
Medium-heavy	15 (05%)	85 (30%)	188 (65%)
High-heavy	107 (37%)	99 (34%)	82 (28%)
Total	351 (30%)	447 (39%)	354 (31%)

Observation 6.12. *The poorest results are generally seen with per-DAG utilizations in the medium-heavy range.* Figures 6.27 and 6.31—scenarios for medium-heavy and high-heavy DAGs, respectively—both show poorer results than the scenarios with parameters that are identical apart from per-DAG utilization. Table 6.2 shows the number of scenarios with each per-task utilization range with RCC values in the good, moderate, and poor ranges.

The two lighter utilization ranges are well-situated to take advantage of SMT: SMT allows two light tasks with combined $U > 1$ to share a core and low-heavy tasks will likely have $U > 1$ after SMT is applied, making them light. High-heavy scenarios also tend to do well; high-heavy tasks are large enough that utilization decreases will generally translate to core count decreases.

Figures 6.31 and 6.32 both show very high RU values. Recall that a DAG's utilization cannot exceed $\frac{C}{L}$; when $\frac{C}{L}$ is less than 4.0 (for medium-heavy DAGs) or 8.0 (for high-heavy DAGs), the DAG will often be assigned a utilization close to $\frac{C}{L}$ and a deadline close to the DAG's length. Because SMT tends to increase DAG length and length can be at most the deadline, there is relatively little opportunity for SMT to be used. Even so, Figure 6.32, depicting high-heavy utilization, does relatively well in terms of RCC, with 0.84, most due to only a small utilization decrease sufficing to reduce core count.

Observation 6.13. *SMT tends to be more helpful to systems of DAGs than to individual DAGs.* When considering the 288 individual DAG scenarios defined using the same criteria as our federated scheduling scenarios, 11% were considered good, 57% were considered moderate, and 31% considered poor. In our federated scheduling experiments, as shown in Table 6.2, 30% of scenarios were considered good, 39% moderate, and 31% poor.

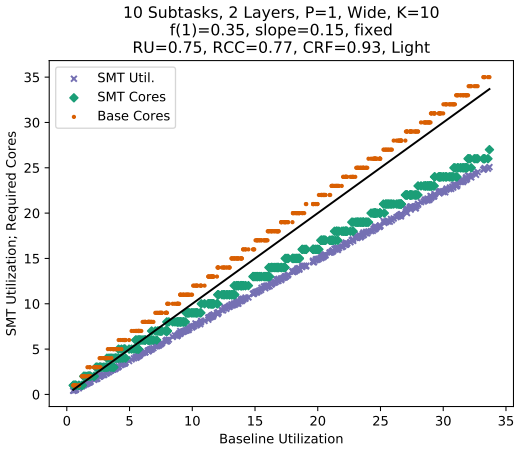

Figure 6.25: Good federated scheduling, light per-dag utilization.

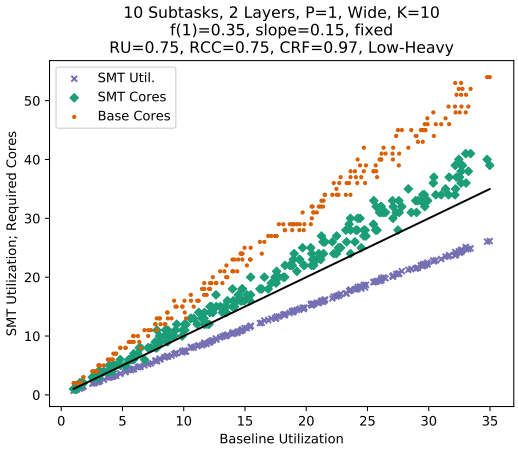

Figure 6.26: Good federated scheduling, low-heavy per-dag utilization.

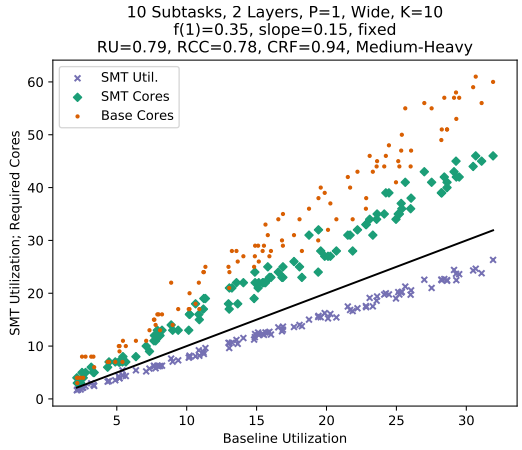

Figure 6.27: Good federated scheduling, medium-heavy per-dag utilization.

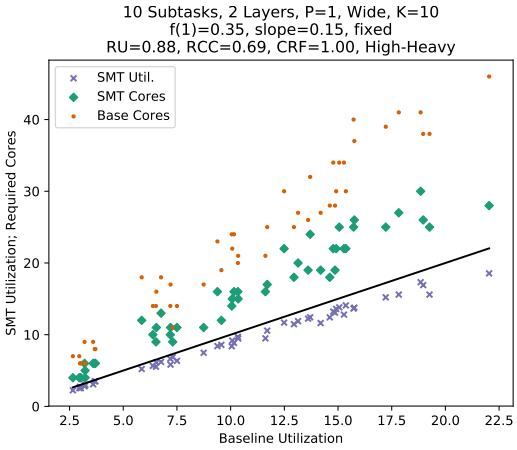

Figure 6.28: Good federated scheduling, high-heavy per-dag utilization.

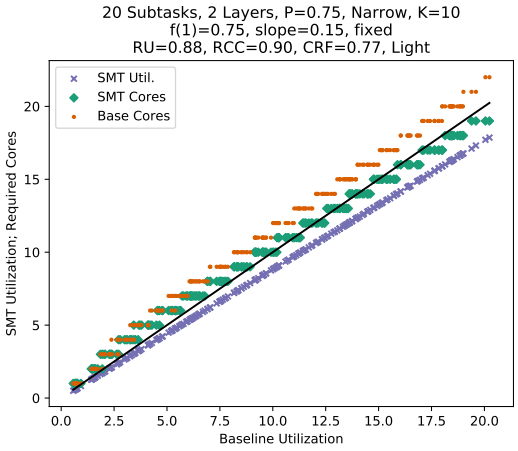

Figure 6.29: Poor federated scheduling, light per-dag utilization.

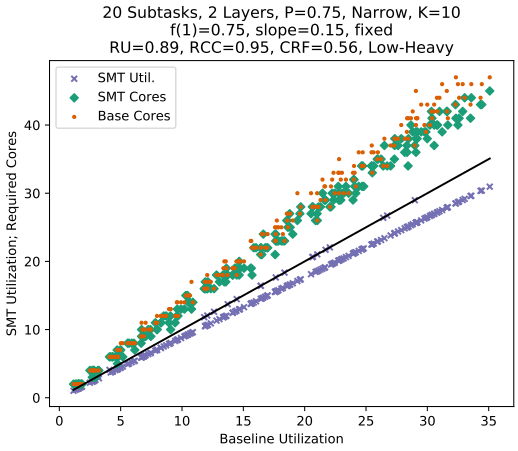

Figure 6.30: Poor federated scheduling, low-heavy per-dag utilization.

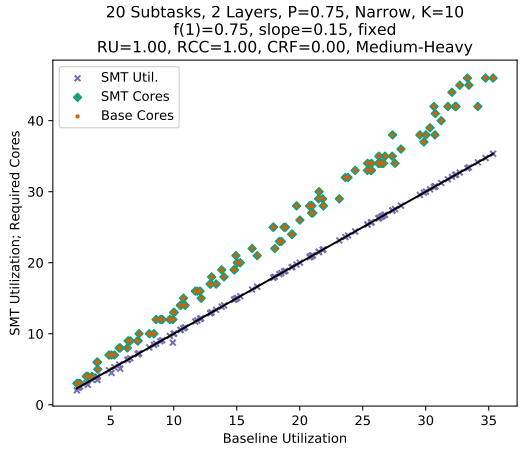

Figure 6.31: Poor federated scheduling, medium-heavy per-dag utilization.

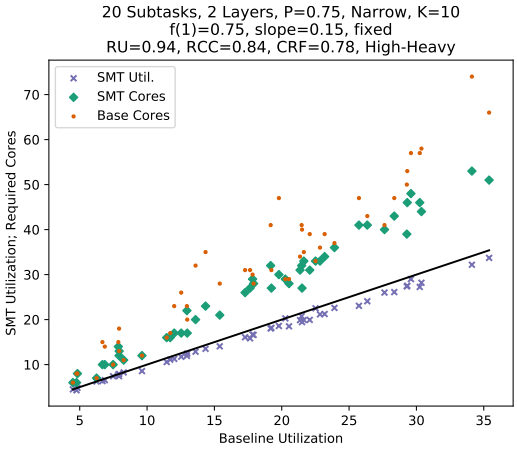

Figure 6.32: Poor federated scheduling, high-heavy per-dag utilization.

6.3 Conclusions and Future Work

In this chapter, we examined the use of SMT to reduce DAG utilization and core count requirements. To reduce the utilization of DAG tasks, we defined an optimization program to apply SMT to DAGs that can minimize DAG utilization. By minimizing utilization, we also reduce core count, albeit non-optimally.

To test our algorithm, we simulated over 70 thousand DAGs across thousands of scenarios. We found that, within our tested scenarios, SMT can reduce DAG utilization and core requirements to 70% or less of what it would have been without SMT.

In addition to evaluating the effects of SMT on individual DAGs, we also applied our methods to systems of DAGs using federated scheduling. We found that systems of DAGs generally saw greater improvement than did individual DAGs with similar per-DAG parameters.

In the future, we hope to expand develop algorithms better-suited for large DAGs and search for core-reduction heuristics other than minimizing utilization.

CHAPTER 7: CONCLUSION

In this chapter we summarize our results, briefly describe additional work we have done regarding SMT, and give directions for future work.

7.1 Summary of Results

In this section we summarize our primary contributions: comparing execution times with and without SMT, using SMT to support SRT systems, and using SMT to support HRT systems.

SMT and execution times. In Chapter 3, we compared execution times of benchmark tasks—we used the TACLeBench sequential benchmarks (Falk et al., 2016)—with and without SMT under conditions appropriate for both SRT and HRT. In both cases, we have found that the increase in execution time from SMT can be as little as 1% and rarely exceeds 100%. These observations tell us that SMT has the potential to improve real-time scheduling, at least in the world of our benchmark choices and execution conditions. Also for both cases, we gave methods for creating systems of synthetic tasks that will exhibit patterns of behavior similar to what we observed.

For HRT timing analysis, we defined the rules of simultaneous co-scheduling, which we use to minimize the variations in timing caused by SMT. In addition, we compared our ability to predict execution times with and without SMT. We found that our approach to timing analysis worked as well with SMT as without. This result suggests that SMT, with proper precautions in place, should be considered as safe for HRT as multicore scheduling.

Using SMT to support SRT systems. In Chapter 4, we gave our method for using SMT to support SRT systems. We decided to schedule as if every SMT-provided scheduling thread were a separate core. Under this approach, any task using SMT can be co-scheduled with any other task using SMT. However, the effects of allowing any one task to use SMT cannot be determined without first deciding which other tasks can use SMT. To solve this circular problem, we created four heuristic algorithms to determine which tasks should and should not use SMT. Once this decision has been made, we can test whether the task system, now divided into two subsystems, can be scheduled using a variation of EDF.

To evaluate our work, we tested over 200 scenarios. We saw some benefit from SMT in almost all cases. In the best cases, SMT could have benefits similar to increasing a platform's core count by 50%.

Using SMT to support sequential HRT systems. In Chapter 5, we gave two algorithms for applying SMT to systems of sequential HRT tasks. Both methods relied on simultaneous co-scheduling to minimize the variation in execution times caused by SMT, and both used an optimization program to determine a good way to use SMT.

Our first approach, CERT-MT, was built around using SMT to combine individual jobs in the context of a CE scheduler. CERT-MT has both the advantages and disadvantages of a CE scheduler. The main advantage is that performance is highly predictable, and the main disadvantage is that the schedule is very rigid; any change to the underlying system requires that the entire scheduled be re-computed. Under favorable circumstances, CERT-MT can effectively increase core counts by 50%, but it suffers from a lack of scalability. While 60 seconds seemed to be sufficient time to compute schedules for the smallest systems, even 300 seconds per system was not necessarily enough to produce good solutions for moderately sized systems.

However, this disadvantage may be less of a problem in practice than it was in our study: while 300 seconds per system is prohibitive in a schedulability study that needs to test thousands of synthetic systems, even a multi-hour runtime may be reasonable when only one actual system needs to be scheduled.

Motivated largely by CERT-MT's scalability problems, we produced a second, priority-driven algorithm for scheduling HRT systems. This approach largely eliminates CERT-MT's scalability problems by using SMT to combine *tasks*, rather than *jobs*, and requiring that only tasks sharing a common period can be paired together. The advantage is a much-improved ability to schedule large systems. The disadvantage is that scheduling systems with few common periods or with more periods than cores (when requiring non-preemptive execution) becomes extremely difficult. Again, in the best cases core count was effectively increased by 50%.

Using SMT to support HRT DAG tasks. In Chapter 6, we showed how to use SMT to support DAG tasks, where portions of individual tasks can be scheduled in parallel. We use SMT to combine individual vertices of a DAG and then schedule the resulting DAG using existing DAG-scheduling algorithms. We found that when using this approach, the number of cores needed to schedule a given DAG could be reduced by as much as 50% under ideal conditions. In addition to evaluating this approach on individual DAGs, we also applied it to systems of DAGs scheduled using Federated Scheduling.

The importance of non-SMT factors. Across Chapters 4, 5, and 6, we saw that whether we used optimistic or pessimistic behaviors for SMT was not the most important factor in determining SMT's effectiveness. In Chapters 4 and 5, the variable that had the most influence on SMT's effectiveness was the utilization ranges of individual tasks and, for the priority-driven scheduler of Chapter 5, the number of periods within a task system. In Chapter 6, the most important variable was usually the degree of parallelism in a DAG before SMT was applied. These observations suggest that our big-picture result—SMT can reduce the number of cores needed to schedule a given system—is not overly reliant on the exact timing behaviors we observed in Chapter 3 or on how we chose to model those behaviors.

7.2 Additional Related Work

In this section we summarize our work on SMT that fell outside the scope of this dissertation.

SMT and mixed-criticality scheduling. *Mixed-Criticality Scheduling* is a scheduling paradigm that allows both HRT and SRT systems to be scheduled on a single hardware platform. Using SMT in this context was the focus of (Bakita et al., 2021). In that paper, we applied earlier versions of our Chapter 3 timing analysis work to two additional benchmark sets; the San Diego Vision Benchmark Suite (SD-VBS) (Venkata et al., 2009) and the Data Intensive Systems stressmarks (Titan Systems Corporation, 2000). These experiments were done on an AMD platform rather than the Intel platform we used in Chapter 3. Our timing analysis conclusions were broadly the same; we found that SMT generally increases execution times by little enough that it is potentially useful for real-time scheduling. In addition, we conducted multiple case studies in which we implemented our schedulers—the SRT scheduler described in Chapter 4 for SRT tasks and the priority-driven scheduler of Section 5.3 for HRT tasks—using real benchmark programs on a real hardware platform. When we did so, we observed no deadline misses for our HRT tasks and no evidence of unbounded tardiness for the SRT tasks. This result indicates that our work holds in practice as well as in theory.

Additional DAG results. Our original paper on applying SMT to DAGs (Osborne et al., 2022), which forms the basis of Chapter 6, included a case study in addition to our schedulability study. We did not include this case study in Chapter 6 as it made use of cache partitioing techniques from (Bakita et al., 2021) that are beyond the scope of our work here. In this case study, we created a series of "semi-synthetic" DAGs where each subtask was defined by a real benchmark program and each DAG was defined by artificially imposing precedence constraints between subtasks. We used the optimization program of Chapter 6 to apply SMT to

these benchmarks. When we executed them using LITMUS^{RT}, a real-time extension of the Linux kernel (Brandenburg, 2011; Calandrino et al., 2006; LITMUS-RT, 2020), we did not observe any deadline misses.

7.3 Future Work

In this section, we give directions for our future work. So far, our work has had more breadth than depth; we have focused on finding different areas in which SMT may be useful at the expense of exploring any one use case in greater detail. In the future, we plan to go into greater detail on the topics we have introduced.

Future timing analysis work. For both SRT and HRT tasks, we observed that execution times were occasionally reduced with SMT. In the future, we want to look more deeply into this behavior with the aim of understanding it better and finding ways to take advantage of this "positive interference." In addition, we would like to take a more code- and hardware-based approach to understanding SMT's effects; our work so far has been entirely measurement-based. Finally, we would like to generalize our models for interference to apply to cross-core interference on multicore platforms rather than SMT alone.

Future SRT scheduling work. Our scheduler of Chapter 4 allows any task using SMT to execute on any core using SMT and any task not using SMT to execute on any core not using SMT. In the future, we would like test the benefits of this approach compared to restricting what cores each task can execute on so as to preserve cache affinity. In addition, we would like to study the ability of SMT to decrease maximum tardiness rather than simply guaranteeing bounded tardiness.

More effective HRT scheduling algorithms. All of our HRT scheduling algorithms can be improved. For CERT-MT, we would like to explore heuristic alternatives that can allow it to handle large task systems. For the priority-driven scheduler of Sections 5.3 and 5.4, we would like to see improvements to its ability to schedule non-preemptable tasks. In particular, minimizing utilization may not always be the best choice. We would like to introduce additional criteria for deciding when to use SMT. For example, it might be advisable to not use SMT when doing so would cause a task's execution time to exceed a certain threshold. As for our DAG scheduling algorithm, we would like to develop an approach that is more suited to scheduling DAGs with 100 subtasks or more.

HRT Sporadic Tasks. In our HRT work, we have been assuming that deciding which jobs should be paired can be done as a preliminary, offline step. Unfortunately, that is not always possible. In a *sporadic* system, each task's period gives the *minimum* time between sequential jobs. The actual time between jobs can

be arbitrarily longer than the period, making it impossible to know in advance whether jobs of two specified tasks will be available to execute at the same time. This limitation is significant: in a 2020 survey of industrial practices in real-time systems (Akesson et al., 2020), 47% of respondents reported working on systems that included sporadic activations.

We want to find, for each task in a sporadic system, the set of job invocation times that will maximize response times with the effects of SMT accounted for. By upper-bounding a task's response time when it releases a job at critical instant—the time that will maximize its response time—we would be able to upper-bound its response time in all cases. By applying this analysis to all tasks, we would be able to determine which when an HRT sporadic system using simultaneous co-scheduling could benefit from SMT.

Competition for non-CPU Resources: resource locking and self-suspending tasks. Throughout this Dissertation, we have assumed that the only item a job needs to execute is a computing core or an SMT-provided thread. This assumption is common in real-time analysis, but it is not always realistic. Tasks may need to access additional hardware components, such as graphics units or I/O devices, as well as intangible items such as specific files or memory regions. Anything apart from computing cores and SMT-provided threads needed for execution is a *resource.* We briefly describe two resource related problems that have been addressed in real-time literature elsewhere but have not been considered in connection with SMT.

When the number of tasks that can access a resource simultaneously is limited, the resource must be *locked* before use; for example, writing to a file generally requires locking the file to prohibit other tasks from accessing it mid-write. When a resource is locked by a lower-priority task, a higher priority task that also needs to use the resource may be forced to wait, causing *priority-inversion blocking*. In addition, checking for resource availability, locking, and unlocking a resource all require time. Additional information on resource locking may be found in (Nemitz, 2021).

Should a job need to wait on a locked resource, it may *suspend* itself, i.e., pause its execution and relinquish its core (or SMT thread) until the resource becomes available. A job may also suspend when it must wait on an I/O device or a hardware accelerator such as a GPU. In either case, self-suspension breaks the assumption that an executing job will continue to execute until either it finishes or is preempted. In many cases, determining schedulability in the presence of self-suspending tasks is NP-complete in the strong sense. Further information on self-suspending tasks can be found in (Chen et al., 2019).

In the future, we would like to consider how resource locking and self-suspending tasks may complicate SMT-related scheduling decisions. One possible question: what should be done if one of two simultaneously

co-scheduled job suspends due to required resource being locked? Should the second job suspend as well? Should an idea of "resource-locking compatibility" or "suspension comptability" be added as an additional restriction of which tasks and jobs can be simultaneously co-scheduled? More generally, will our approach to SRT tasks of treating SMT threads as separate cores and mimicking existing multi-core work as closely as possible still work given the need to lock resources or self-suspend? Answering these questions would serve to increase the variety of systems that can benefit from SMT.

BIBLIOGRAPHY

Akesson, B., Nasri, M., Nelissen, G., Altmeyer, S., and Davis, R. (2020). An empirical survey-based study into industry practice in real-time systems. In *Proceedings of the 41st IEEE Real-Time Systems Symposium*.

Altmeyer, S., Douma, R., Lunniss, W., and Davis, R. (2016). On the effectiveness of cache partitioning in hard real-time systems. *Real-Time Systems*, 52:598–643.

Altmeyer, S., Douma, R., Lunniss, W., and Davis, R. I. (2014). Outstanding paper: Evaluation of cache partitioning for hard real-time systems. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems*.

Amert, T., Balszun, M., Geier, M., Smith, F. D., Anderson, J., and C., S. (2021). Timing-predictable vision processing for autonomous systems. In *Proceedings of the 24th Design, Automation, and Test in Europe Conference and Exhibition*.

Anantaraman, A., Seth, K., Patil, K., Rotenberg, E., and Mueller, F. (2003). Virtual simple architecture (VISA) exceeding the complexity limit in safe real-time systems. In *Proceedings of the 30th annual International Symposium on Computer architecture*.

Arcaro, L. F., Silva, K. P., de Oliveira, R. S., and Almeida, L. (2020). Reliability test based on a binomial experiment for probabilistic worst-case execution times. In *Proceedings of the 41st IEEE Real-Time Systems Symposium*.

Baker, T. P. and Shaw, A. (1989). The cyclic executive model and Ada. *Real-Time Systems*, 1(1):7–25.

Bakita, J., Osborne, S., Ahmed, S., Tang, S., Chen, J., Smith, F., and Anderson, J. (2021). Simultaneous multithreading in mixed-criticality real-time systems. In *Proceedings of the 27th IEEE Real-Time Technology and Applications Symposium*.

Balkema, A. and De Haan, L. (1974). Residual life time at great age. *The Annals of Probability*, pages 792–804.

Baruah, S. (2015). The federated scheduling of constrained-deadline sporadic DAG task systems. In *Proceedings of the Design, Automation, and Test in Europe Conference and Exhibition*.

Baruah, S., Bonifaci, V., Marchetti-Spaccamela, A., Stougie, L., and Wiese, A. (2012). A generalized parallel task model for recurrent real-time processes. In *Proceedings of the 33rd IEEE Real-Time Systems Symposium*.

Baruah, S., Cohen, N., Plaxton, C., and Varvel, D. (1993). Proportionate progress: A notion of fairness in resource allocation. In *Proceedings of the 25th annual ACM symposium on Theory of Computing*.

Bernat, G., Burns, A., and Liamosi, A. (2001). Weakly hard real-time systems. *IEEE Transactions on Computers*, 50(4):308–321.

Brandenburg, B. (2011). *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, UNC Chapel Hill. https://www.cs.unc.edu/ anderson/diss/bbbdiss.pdf.

Bui, B. D., Caccamo, M., Sha, L., and Martinez, J. (2008). Impact of cache partitioning on multi-tasking real time embedded systems. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*.

Bulpin, J. (2005). *Operating system support for simultaneous multithreaded processors*. PhD thesis, University of Cambridge, King's College. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-619.pdf.

Bulpin, J. and Pratt, I. (2004). Multiprogramming performance of the Pentium 4 with hyperthreading. In *Third Annual Workshop on Duplicating, Deconstruction and Debunking*.

Burns, A. and Baruah, S. (2017). Migrating mixed criticality tasks within a cyclic executive framework. In *Reliable Software Technologies – Ada-Europe 2017*, pages 203–216, Cham. Springer International Publishing.

Burns, A. and Edgar, S. (2000). Predicting computation time for advanced processor architectures. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*.

Burns, A., Fleming, T., and Baruah, S. (2015). Cyclic executives, multi-core platforms and mixed criticality applications. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems*.

Calandrino, J. M., Leontyev, H., Block, A., Devi, U. C., and Anderson, J. (2006). LITMUS^{RT} : A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 111–126.

Carpenter, J., Funk, S., Holman, P., Srinivasan, A., Anderson, J., and Baruah, S. (2004). A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca.

Cazorla, F. J., Knijnenburg, P. M. W., Sakellariou, R., Fernandez, E., Ramirez, A., and Valero, M. (2006). Predictable performance in SMT processors: synergy between the OS and SMTs. *IEEE Transactions on Computers*, 55(7):785–799.

Cazorla, F. J., Kosmidis, L., Mezzetti, E., Hernandez, C., Abella, J., and Vardanega, T. (2019). Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey. *ACM Computing Surveys*, 52(1):14:1–14:35.

Chen, J.-J., Nelissen, G., Huang, W.-H., Yang, M., Brandenburg, B., Bletsas, K., Liu, C., Richard, P., Ridouard, F., Audsley, N., et al. (2019). Many suspensions, many problems: a review of self-suspending tasks in real-time systems. *Real-Time Systems*, 55(1):144–207.

Chisholm, M., Kim, N., Tang, S., Otterness, N., Anderson, J., Smith, F. D., and Porter, D. (2017). Supporting mode changes while providing hardware isolation in mixed-criticality multicore systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*.

Chisholm, M., Ward, B. C., Kim, N., and Anderson, J. (2015). Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *Proceedings of the IEEE Real-Time Systems Symposium*.

Choi, H., Kim, H., and Zhu, Q. (2021). Toward practical weakly hard real-time systems: A job-class-level scheduling approach. *IEEE Internet of Things Journal*, 8(8):6692–6708.

Cordeiro, D., Mounié, G., Perarnau, S., Trystram, D., Vincent, J.-M., and Wagner, F. (2010). Random graph generation for scheduling simulations. In *3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools 2010)*, page 10. ICST.

Davis, R. and Cucu-Grosjean, L. (2019). A survey of probabilistic timing analysis techniques for real-time systems. *Leibniz Transactions on Embedded Systems*, 6(1):03–1–03:60.

Deutschbein, C., Fleming, T., Burns, A., and Baruah, S. (2019). Multi-core cyclic executives for safety-critical systems. *Science of Computer Programming*, 172:102 – 116.

Devi, U. and Anderson, J. (2005). Tardiness bounds under global EDF scheduling on a multiprocessor. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*.

Devi, U. and Anderson, J. (2006). Flexible tardiness bounds for sporadic real-time task systems on multiprocessors. In *20th IEEE International Parallel Distributed Processing Symposium*.

Dhall, S. and Liu, C. (1978). On a real-time scheduling problem. *Operations Research*, 26(1):127–140.

Dorai, G. K., Yeung, D., and Choi, S. (2003). Optimizing SMT processors for high single-thread performance. Technical report.

Eggers, S., Emer, J., Levy, H., Lo, J., Stamm, R., and Tullsen, D. (1997). Simultaneous multithreading: a platform for next-generation processors. *IEEE Micro*, 17(5):12–19.

Ekberg, P. and Baruah, S. (2021). Partitioned scheduling of recurrent real-time tasks. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium*.

Elliott, G., Yang, K., and Anderson, J. (2015). Supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms. In *Proceedings of the 36th IEEE Real-Time Systems Symposium*.

Erdős, P. and Rényi, A. (1960). On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 5(1):17–60.

Falk, H., Altmeyer, S., Hellinckx, P., Lisper, B., Puffitsch, W., Rochange, C., Schoeberl, M., Sørensen, R. B., Wägemann, P., and Wegener, S. (2016). TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis*.

Fisher, R. A. and Tippett, L. H. C. (1928). Limiting forms of the frequency distribution of the largest or smallest member of a sample. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 24, pages 180–190. Cambridge University Press.

Fog, A. (2018). The microarchitecture of Intel, AMD, and VIA CPUs: an optimization guide for assembly programmers and compiler makers. Available at `https://www.agner.org/optimize/microarchitecture.pdf`.

Gomes, T., Garcia, P., Pinto, S., Monteiro, J., and Tavares, A. (2016). Bringing hardware multithreading to the real-time domain. *IEEE Embedded Systems Letters*, 8(1):2–5.

Gomes, T., Pinto, S., Garcia, P., and Tavares, A. (2015). RT-SHADOWS: Real-time system hardware for agnostic and deterministic OSes within softcore. In *Proceedings of the 20th IEEE International Conference on Emerging Technologies and Factory Automation*.

Graham, R. (1969). Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429.

Guo, D. and Pellizzoni, R. (2017). A requests bundling DRAM controller for mixed-criticality systems. In *Proceedings of the 23rd IEEE Real-Time Technology and Applications Symposium*.

Guo, Z., Bhuiyan, A., Liu, D., Khan, A., Saifullah, A., and Guan, N. (2019). Energy-efficient real-time scheduling of DAGs on clustered multi-core platforms. In *Proceedings of the 25th IEEE Real-Time Technology and Applications Symposium*.

Guo, Z., Yang, K., Yao, F., and Awad, A. (2020). Inter-task cache interference aware partitioned real-time scheduling. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*.

Gurobi Optimization LLC (2023). Gurobi Optimizer Reference Manual.

Hammadeh, Z. A. H., Quinton, S., and Ernst, R. (2019). Weakly-hard real-time guarantees for earliest deadline first scheduling of independent tasks. *ACM Transactions on Embedded Computing Systems*, 18(6).

Hassan, M., Patel, H., and Pellizzoni, R. (2015). A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In *Proceedings of the 21st IEEE Real-Time Technology and Applications Symposium*.

He, Q., Guan, N., Lv, M., Jiang, X., and Chang, W. (2022). Bounding the response time of DAG tasks using long paths. In *Proceedings of the 43rd IEEE Real-Time Systems Symposium*.

Hennessy, J. L. and Patterson, D. A. (2019). *Computer architecture: a quantitative approach (sixth edition)*. Morgan Kaufmann Publishers.

Houssam-Eddine, Z., Capodieci, N., Cavicchioli, R., Lipari, G., and Bertogna, M. (2021). The HPC-DAG task model for heterogeneous real-time systems. *IEEE Transactions on Computers*, 70(10):1747–1761.

Houssam-Eddine, Z., Lipari, G., Bertogna, M., and Boulet, P. (2019). The parallel multi-mode digraph task model for energy-aware real-time heterogeneous multi-core systems. *IEEE Transactions on Computers*, 68(10):1511–1524.

Huang, C., Li, W., and Zhu, Q. (2019). Formal verification of weakly-hard systems. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*.

Jain, R., Hughes, C., and Adve, S. (2002). Soft real-time scheduling on simultaneous multithreaded processors. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*.

Jeffay, K., Stanat, D., and Martel, C. (1991). On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the 12th IEEE Real-Time Systems Symposium*.

Jiang, X., Guan, N., and Long, X.and Yi, W. (2017). Semi-federated scheduling of parallel real-time tasks on multiprocessors. In *Proceedings of the 38th IEEE Real-Time Systems Symposium*.

Jiménez Gil, S., Bate, I., Lima, G., Santinelli, L., Gogonel, A., and Cucu-Grosjean, L. (2017). Open challenges for probabilistic measurement-based worst-case execution time. *IEEE Embedded Systems Letters*, 9(3):69–72.

Kato, S., Takeuchi, E., Ishiguro, Y., Ninomiya, Y., Takeda, K., and Hamada, T. (2015). An open approach to autonomous vehicles. *IEEE Micro*, 35(6):60–68.

Kim, H., Kandhalu, A., and Rajkumar, R. (2013). A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*.

Kim, J., Yoon, M., Bradford, R., and Sha, L. (2014). Integrated modular avionics (IMA) partition scheduling with conflict-free I/O for multicore avionics systems. In *2014 IEEE 38th Annual Computer Software and Applications Conference*.

Kirk, D. (1988). Process dependent static cache partitioning for real-time systems. In *Proceedings of the 9th IEEE Real-Time Systems Symposium*.

Kirk, D. (1989). SMART (strategic memory allocation for real-time) cache design. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*.

Leung, J. and Merrill, M. (1980). A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115–118.

Li, J., Agrawal, K., and Lu, C. (2022). Parallel real-time scheduling. In *Handbook of Real-Time Computing*, pages 447–467. Springer.

Li, J., Chen, J. J., Agrawal, K., Lu, C., Gill, C., and Saifullah, A. (2014). Analysis of federated and global scheduling for parallel real-time tasks. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems*.

Li, J., Dinh, S., Kieselbach, K., Agrawal, K., Gill, C., and Lu, C. (2016). Randomized work stealing for large scale soft real-time systems. In *Proceedings of the 37th IEEE Real-Time Systems Symposium*.

Liedtke, J., Hartig, H., and Hohmuth, M. (1997). OS-controlled cache predictability for real-time systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*.

LITMUS-RT (2020). LITMUS^{RT} home page. Online at `http://www.litmus-rt.org/`.

Liu, C. and Layland, J. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61.

Liu, J. W. S. (2000). *Real-Time Systems*. Prentice Hall, New York, NY, USA.

Lo, S., Lam, K., and Kuo, T. (2005). Real-time task scheduling for SMT systems. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*.

Maiza, C., Rihani, H., Rivas, J., Goossens, J., Altmeyer, S., and Davis, R. (2019). A survey of timing verification techniques for multi-core real-time systems. *ACM Computing Surveys*, 52(3).

Mancuso, R., Dudko, R., Betti, E., Cesati, M., Caccamo, M., and Pellizzoni, R. (2013). Real-time cache management framework for multi-core architectures. In *Proceedings of the 19th IEEE Real-Time Technology and Applications Symposium*.

Marr, D., Binns, F., Hill, D., Hinton, G., Koufaty, K., Miller, J., and Upton, M. (2002). Hyper-threading technology architecture and microarchitecture. In *Intel Technology Journal*, volume 6, pages 4–15.

Martel, C. (1982). Preemptive scheduling with release times, deadlines, and due times. *Journal of the ACM*, 29(3):812–829.

Melani, A., Mancuso, R., Caccamo, M., Buttazzo, G., Freitag, J., and Uhrig, S. (2017). A scheduling framework for handling integrated modular avionic systems on multicore platforms. In *Proceedings of the 23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*.

Mills, A. and Anderson, J. (2011). A multiprocessor server-based scheduler for soft real-time tasks with stochastic execution demand. In *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, volume 1.

Mische, J., Uhrig, S., Kluge, F., and Ungerer, T. (2010). Using SMT to hide context switch times of large real-time tasksets. In *Proceedings of the 16th IEEE Real-Time Technology and Applications Symposium*.

Mok, A. (1983). *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology. https://dspace.mit.edu/bitstream/handle/1721.1/15670/10728774-MIT.pdf.

Mueller, F. (1995). Compiler support for software-based cache partitioning. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, Tools for Real-Time Systems*, page 125–133.

Muench, D., Paulitsch, M., and Herkersdorf, A. (2014). Temporal separation for hardware-based I/O virtualization for mixed-criticality embedded real-time systems using PCIe SR-IOV. In *Workshop Proceedings on Architecture of Computing Systems (ARCS)*.

Nemitz, C. (2021). *Efficient Synchronization for Real-Time Systems with Nested Resource Access*. PhD thesis, UNC Chapel Hill. https://cs.unc.edu/ anderson/diss/nemitzdiss.pdf.

Osborne, S., Ahmed, S., Nandi, S., and Anderson, J. (2020). Exploiting simultaneous multithreading in priority-driven hard real-time systems. In *Proceedings of the 26th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*.

Osborne, S. and Anderson, J. (2020). Simultaneous multithreading and hard real time: Can it be safe? In *Proceedings of the 32nd Euromicro Conference on Real-Time Systems*.

Osborne, S., Bakita, J., and Anderson, J. (2019). Simultaneous multithreading applied to real time. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems*.

Osborne, S., Bakita, J., Chen, J., Yandrofski, T., and Anderson, J. (2022). Minimizing DAG utilization by exploiting SMT. In *Proceedings of the 28th IEEE Real-Time Technology and Applications Symposium*.

Patterson, D. and Sequin, C. (1981). RISC I: A reduced instruction set vlsi computer. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, page 443–457, Washington, DC, USA. IEEE Computer Society Press.

Pazzaglia, P., Sun, Y., and Natale, M. D. (2020). Generalized weakly hard schedulability analysis for real-time periodic tasks. *ACM Transactions on Embedded Computing Systems*, 20(1).

Pellizzoni, R., Bui, B. D., Caccamo, M., and Sha, L. (2008). Coscheduling of CPU and I/O transactions in COTS-based embedded systems. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*.

Pickands, J. (1975). Statistical inference using extreme order statistics. *the Annals of Statistics*, 3(1):119–131.

Rapita Systems (2021). A commercial solution for safety-critical multicore timing analysis [white paper]. Available at `https://www.rapitasystems.com/downloads/commercial-solution-safety-critical-multicore-timing-analysis`.

Reder, S. and Becker, J. (2020). Interference-aware memory allocation for real-time multi-core systems. In *Proceedings of the 26th IEEE Real-Time Technology and Applications Symposium*.

Reghenzani, F., Massari, G., Fornaciari, W., and Galimberti, A. (2019). Probabilistic-WCET reliability: On the experimental validation of EVT hypotheses. In *Proceedings of the International Conference on Omni-Layer Intelligent Systems*.

Seetanadi, G., Camara, J., Almeida, L., Arzen, K., and Maggio, M. (2017). Event-driven bandwidth allocation with formal guarantees for camera networks. In *Proceedings of the 40th IEEE Real-Time Systems Symposium*, pages 243–254.

Srinivasan, A. and Anderson, J. (2006). Optimal rate-based scheduling on multiprocessors. *Journal of Computer and System Sciences*, 6(72):1094–1117.

Sun, Y. and Natale, M. (2017). Weakly hard schedulability analysis for fixed priority scheduling of periodic real-time tasks. *ACM Transactions on Embedded Computing Systems*, 16(5s).

Titan Systems Corporation (2000). DIS Stressmark Suite. Technical report. ver. 1.0.

Tobita, T. and Kasahara, H. (2002). A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. *Journal of Scheduling*, 5(5):379–394.

Tuck, N. and Tullsen, D. M. (2003). Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of the 7th Conference on Parallel Architectures and Compilation Techniques*, pages 26–35, Washington, DC, USA.

Tullsen, D., Eggers, S., Emer, J., Levy, H., Lo, J., and Stamm, R. (1996). Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. *SIGARCH Computer Architecture News*, 24(2):191–202.

Tullsen, D., Eggers, S., and H.Levy (1995). Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the International Symposium on Computer Architecture*, pages 392–403.

Ueter, N., Von Der Brüggen, G., Chen, J.-J., Li, J., and Agrawal, K. (2018). Reservation-based federated scheduling for parallel real-time tasks. In *Proceedings of the 39th IEEE Real-Time Systems Symposium*.

Valsan, P., Yun, H., and Farshchi, F. (2016). Taming non-blocking caches to improve isolation in multicore real-time systems. In *Proceedings of the 22nd IEEE Real-Time Technology and Applications Symposium*.

Venkata, S., Ahn, I., Jeon, D., Gupta, A., Louie, C., Garcia, S., Belongie, S., and Taylor, M. (2009). SD-VBS: the San Diego vision benchmark suite. In *IISWC*, pages 55–64.

Ward, B., Herman, J., Kenna, C., and Anderson, J. (2013). Outstanding paper award: Making shared caches more predictable on multicore platforms. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*.

Wilhelm, R. (2020). Real time spent on real time (invited talk). In *Proceedings of the 41st IEEE Real-Time Systems Symposium*.

Wolfe, A. (1993). Software-based cache partitioning for real-time applications. In *Proceedings of the 3rd International Workshop on Responsive Computer Systems*.

Xu, M., Phan, L., Choi, H., and Lee, I. (2016). Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation. In *Proceedings of the 22nd IEEE Real-Time Technology and Applications Symposium*.

Xu, M., Phan, L., Choi, H., Lin, Y.and Li, H., Lu, C., and Lee, I. (2019). Holistic resource allocation for multicore real-time systems. In *Proceedings of the 25th Real-Time and Embedded Technology and Applications Symposium*.

Yang, K., Elliott, G., and Anderson, J. (2015). Analysis for supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms. In *Proceedings of the 23rd International Conference on Real-Time Networks and Systems*.

Ye, Y., West, R., Cheng, Z., and Li, Y. (2014). COLORIS: A dynamic cache partitioning system using page coloring. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*.

Yun, H., Mancuso, R., Wu, Z., and Pellizzoni, R. (2014). PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Proceedings of the 20th IEEE Real-Time Technology and Applications Symposium*.

Yun, H., Yao, G., Pellizzoni, R., Caccamo, M., and Sha, L. (2013). Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Proceedings of the 19th IEEE Real-Time Technology and Applications Symposium*.

Zimmer, M., Broman, D., Shaver, C., and Lee, E. A. (2014). FlexPRET: A processor platform for mixed-criticality systems. In *Proceedings of the 2014 IEEE Real-Time Technology and Applications Symposium*.