Designing Predictable Time-Aware and Energy-Efficient Cyber-Physical Systems

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Sandeep Maurice D'souza

B. Tech, Indian Institute of Technology Kharagpur

Carnegie Mellon University
Pittsburgh, PA

December 2019

Abstract

Cyber-physical systems (CPS) involve the *cyber* components of computing and communication interacting with and controlling elements in the *physical* world. Emerging CPS are increasingly distributed and perform coordinated sensing and actuation over large geographical areas. Examples include local-scale industrial robots, city-scale traffic management, and regional/continental-scale smart grids. Hence, a hierarchy of resource-constrained embedded sensing/actuation nodes, edge cloudlets and the cloud will be key to enable scalable coordination, while simultaneously hosting the *intelligence* behind these systems. To meet the low-latency *real-time* requirements of CPS, these platforms typically harness a variety of computing resources ranging from multi-core processors to hardware accelerators such as general-purpose Graphics-Processing Units (GP-GPUs).

In conjunction with low latency, a shared and precise notion of time is key to enabling coordinated action in distributed CPS. Hence, in this dissertation, we introduce abstractions, system-design methodologies and frameworks that enable time-based coordination in geo-distributed cyber-physical systems. While a shared notion of time enables coordination at the *distributed* scope, to coordinate effectively it is also necessary to simultaneously schedule multiple application components at the scope of each *node*, such that all deadlines are met, while ensuring that the resource/physical constraints of the system are satisfied. Therefore, this dissertation also introduces energy-, thermal- and resource-efficient analyzable real-time scheduling techniques for applications deployed on platforms utilizing both multi-core processors and hardware accelerators.

Our proposed solutions are readily applicable to commodity embedded, edge and cloud platforms, and together can enable time-aware and energy-efficient CPS.

To my parents: Errol and Meera D'souza

Acknowledgements

I am deeply grateful to all the individuals who have contributed to enriching my PhD experience. This dissertation would not have been possible without their support.

First and foremost, I would like to thank my advisor, Professor Raj Rajkumar. I am grateful for the multiple opportunities which Raj provided, to work on a diverse set of projects, ranging from theoretical real-time scheduling, to distributed cyber-physical systems and autonomous vehicles. Working with Raj has been a privilege and provided multiple avenues for personal growth. His guidance and advice have positively influenced my approach to research. In particular, his big-picture-driven top-down approach to problem solving, and his focus on the importance of both theoretical contributions and real-world system building have definitely made me both a better researcher and engineer. Raj's guidance and constructive criticism have also helped me improve my technical writing and presentation skills.

I am grateful to the members of my thesis committee, Professor Aniruddha Gokhale, Professor Anthony Rowe and Professor Mani Srivastava for taking the time and effort to be a part of this process. Their valuable comments and suggestions played a key role in shaping up this dissertation. I am especially grateful to Professors Anthony and Mani. Collaborating with them and their respective research groups has been an enriching experience. Additionally, interacting with both Anthony and Mani has helped me broaden my perspectives on both research and life. It has also been a pleasure interacting with Professor Aniruddha. His approach to research and his friendly nature are aspects that I would like to emulate.

A special word of gratitude to the National Science Foundation (NSF) for funding my research, as part of the Roseline project (CNS-1329644). Being a part of this project provided me an opportunity to solve challenging problems, as well as collaborate with a group of smart people. In particular, I would like to thank Fatima Anwar, Adwait Dongare, Andrew Symington and Anh Luong. Collaborating with them provided a

solid foundation on which I could develop my own research ideas. In addition, the yearly trips to Washington DC for the project review provided interesting experiences to cherish for a lifetime.

My internship experiences have played an important role in my journey as a graduate student. For this, I am grateful to all my internship managers for providing these opportunities. My sincere thanks to Hasan Sinan Bank for introducing me to the world of robotics and broadening my research perspectives. Sinan's strong work ethic and approach to problem solving has left a lasting impact. A special word of thanks to Akhilesh Joshi, Heiko Koehler and Satyam Vaghani. They have been a great source of inspiration and support in the course of my PhD journey. Working with them introduced me to challenges involved in designing software for the real world, and provided me a holistic view of how industry functions. I am also grateful to Landon Cox and Victor Bahl for providing me the unique opportunity to work at the intersection of privacy and video analytics, despite my lack of prior experience in the domain. Interacting with Landon and Victor has shown me the importance of seizing opportunities as and when they arise.

An important part of my PhD journey has been being a member of the Real-Time and Multimedia systems Lab (RTML). During my time at RTML, I have had the opportunity of interacting with many wonderful people including Hyoseung Kim, Anand Bhat, Shunsuke Aoki, Iljoo Baek, Mengwen He, Weijing Shi, Peter Jan and Swapnil Das. I would like to thank all of them for their support, and for fostering a collaborative and collegial atmosphere. I cherish the friendships and research collaborations I have fostered as a result of my time at RTML. I would also like to thank Toni Fox, Chelsea Mendenhall, Brittany Frost, Brigette Bernagozzi and Nathan Snizaski for their kind support on administrative matters.

I am privileged to have had a good group of friends in the course of my student life: Sanghamitra Dutta, Ankur Mallick, Akshay Gadre, Samarth Gupta, Rajshekhar Das, Rajat Kateja, Sayan Choudhury, Esha Chatterjee, Mansi Sood, Abhilasha Jain, Deepoo

Kumar, Rohit Singh, Nirjhar Bera and Reekhiya Basu. In particular, my fiancée Sanghamitra has been a great source of encouragement and support for the past eight years. Her kind and loving nature have always kept me going through the ups and downs of life. I am also grateful to my undergraduate and high-school friends who have kept in touch through the years: Shivam Saxena, Bibek Ranjan Sahu, Aashish Kumar, Prateek Vashishtha, Sanchari Sen, Deepika Banoth, Tanvi Ranjan, Anit Sahu, Diwakar Paliwal and Harshawardhan Gupta. A special word of gratitude to Anit, who has been a great mentor since my undergraduate days.

Finally, I would like to thank my family. I am grateful to my parents Errol and Meera D'souza, and my younger brother Aditya D'souza, for believing in me, and always encouraging me to pursue my dreams. I am also indebted to my grandparents, especially my Nani (maternal grandmother) for always being there for me. My family have always been there to guide and support me, and I dedicate this thesis to them.

Contents

Abstract iii

Acknowledgements v

Contents **viii**

List of Tables **xi**

List of Figures **xii**

1 Introduction **1**

1.1 Motivation 1

1.2 Scope of the Thesis 5

1.3 Contributions 13

1.4 Organization 18

2 Related Work **20**

2.1 Time-based Distributed Coordination 20

2.2 Energy-Aware Multi-Core Real-Time Scheduling 24

2.3 Real-Time Scheduling for Hardware Accelerators 28

3 Enabling Time-based Coordination in Cyber-Physical Systems **31**

3.1 The Case for Shared Time and QoT 33

3.2 Timelines 36

3.3 QoT Architecture and Stack 39

3.4 Clocks . . . 40
3.5 System Services . . . 41
3.6 QoT Core . . . 43
3.7 Application Programming Interface . . . 44
3.8 Experimental Evaluation . . . 46
3.9 Summary . . . 53

4 Bringing QoT to Virtual Machines 55
4.1 Background . . . 56
4.2 Time-Based Applications using QoT . . . 59
4.3 QuartzV Extension to the QoT Stack . . . 62
4.4 Experimental Evaluation . . . 72
4.5 Summary . . . 86

5 Time-as-a-Service for Geo-distributed Coordination 87
5.1 An Application's Perception of Time . . . 88
5.2 Quartz: *Time-as-a-Service* (TaaS) . . . 91
5.3 Evaluation . . . 107
5.4 Summary . . . 121

6 Sleep Scheduling for Energy-Savings in Multi-Core Processors 123
6.1 Energy-Saving Rate-Harmonized Scheduling . . . 124
6.2 Energy-Saving Rate-Monotonic Scheduling . . . 130
6.3 Energy-Saving Schedulers . . . 131
6.4 Sleep-State based Energy-Saving on Multi-Core Processors . . . 132
6.5 SyncSleep Scheduling . . . 134
6.6 IndSleep Scheduling . . . 142
6.7 Comparative Evaluations . . . 143
6.8 Summary . . . 147

7 Thermal Implications of Energy-Saving Schedulers 149

7.1 Thermal Modeling of ES Schedulers 150

7.2 SysSleep Algorithm 153

7.3 Thermal-Aware Multi-Core ES Scheduling 160

7.4 Comparative Evaluation 169

7.5 Summary 175

8 Energy-Saving Scheduling for Real-Time Systems with Hardware Accelerators 177

8.1 System Model 178

8.2 CycleSolo Algorithm 183

8.3 CycleTandem Algorithm 194

8.4 Multi-core CycleSolo and CycleTandem 199

8.5 Experimental Evaluation 205

8.6 Summary 212

9 Co-Scheduling Real-Time Workloads on Concurrent Hardware Accelerators 214

9.1 Background and System Model 215

9.2 Work-Conserving Fixed-Priority Scheduling 218

9.3 Non-Work-Conserving FIFO Scheduling 239

9.4 Concurrent Accelerator Partitioning 240

9.5 Comparative Evaluation 245

9.6 Summary 251

10 Conclusions and Future Work 252

10.1 Future Directions 256

Bibliography 259

List of Tables

3.1 Quality of Time APIs 44

4.1 Clock-Read Latency (nanoseconds) 78

5.1 NTP [1] Accuracy Micro-benchmarks (μseconds) 109
5.2 PTP [2] Accuracy Micro-benchmarks (μseconds) 110
5.3 Huygens [3] Accuracy Micro-benchmarks (μs) 112
5.4 Continent-scale Scalability Results 118
5.5 Geo-distributed Scalability Results: Microsoft Azure 119
5.6 Geo-distributed Scalability Results: Google Cloud 120

8.1 Experimental Tasksets Deployed on the NVIDIA TX2 210
8.2 NVIDIA TX2 Taskset Parameters (ms) 211
8.3 CPU,GPU Frequencies used on the NVIDIA TX2 (MHz) 211
8.4 Power Measurements on the NVIDIA TX2 (milliwatts) 212

List of Figures

1.1 The scale of coordination in time and space 2

1.2 *DronePorter*: Time-based Drone coordination 3

1.3 Task τ_1 executing using a combination of CPU and GPU resources 11

1.4 Tasks τ_1 and τ_2 executing concurrently on the GPU 12

3.1 Coordinating subgroups in a cyber-physical system require access to a shared sense of time 32

3.2 Hierarchy of the dynamic traffic-management application. The text on the left highlights the components, while the text on the right indicates the networking technology used to connect adjacent levels of the hierarchy 34

3.3 Timeline-driven Quality-of-Time Architecture 36

3.4 Traditional v/s Timeline-based synchronization models [4] 38

3.5 The QoT Stack for Linux 40

3.6 Timeline-based end-to-end time synchronization 42

3.7 (a) Core-NIC synchronization accuracy (b) Illustrating the adjustable synchronization parameter 47

3.8 (a) shows pair-wise error probability density of three nodes a, b, c bound to Timeline 1 in Figure 3.4b with 100 μsec accuracy requirement, while (b) shows pair-wise error probability density of three nodes c, d, e bound to Timeline 2 with 1 μsec accuracy (Note that x-axis units are in nanoseconds, and x-axis scale changes in (a) and (b)). Note that c maintains mappings of both timelines, and the achieved accuracy for all the nodes is almost equal to their desired accuracy 48

3.9 Upper bound (upper green plot) and lower bound (lower blue plot) around the actual uncertainty (middle red plot) with and without synchronization. Note the change in y-axis scale which is increasing from (a) to (c) 49

3.10 Scheduler Latency Distributions, for a periodic pin-toggling application on a single node 51

3.11 End-to-End Scheduling Jitter Distributions, for a distributed synchronous pin-toggling application deployed on two nodes 52

3.12 Clock read latency histograms in different time intervals, estimated by the system uncertainty estimation service 53

4.1 The QEMU-KVM Hypervisor. VM 1 supports para-virtual peripheral access, while VM N utilizes peripheral emulation (full virtualization) 58

4.2 Time-based Coordinated Industrial Automation 62

4.3 Specifying QoT information from guest applications to host service `qot_virtd` using VirtIO serial 68

4.4 Sharing clock parameters and QoT information from host service `qot_virtd` to guest VM applications using `ivshmem` 69

4.5 QoT Stack for Linux in a fully-virtualized guest VM 70

4.6 QuartzV Clock-Synchronization Test-bed 72

4.7 Measured Clock-Synchronization Error Distributions. The y-axis represents the probability density, and the x-axis the measured error 75

4.8 Clock-Synchronization Accuracy Boxplot. The center 'red' line represents the median accuracy, the inner whiskers the 25th and 75th percentile accuracy, and the outer whiskers the minimum and maximum error observed. 76
4.9 QoT Bounds: (a) para-virtual QuartzV, (b) fully-virtual QoT Stack 77
4.10 QuartzV Synchronization Scalability Results. The dashed lines represent moments in time where a new VM was spawned 79
4.11 Fully-Virtual QoT Stack Synchronization Scalability Results. The dashed lines represent moments in time where a new VM was spawned 80
4.12 QuartzV Synchronization Scalability Results with per-VM Network Reception/Transmission Bandwidth restricted to 2 MB/s 83
4.13 Fully-Virtual QoT Stack Synchronization Scalability Results with per-VM Network Reception/Transmission Bandwidth restricted to 2MB/s 84
4.14 QuartzV Clock-Read Scalability Results 85

5.1 *TimeCop*: City-Scale Traffic Management 91
5.2 Quartz Time-as-a-Service. Solid boxes indicate components, and dashed boxes indicate interfaces. 95
5.3 Quartz Time-as-a-Service at global scope 99
5.4 Adaptive NTP: Server Selection & Rate Adaptation 104
5.5 Quartz NTP: (a) Adaptive clock-synchronization, (b) QoT bounds on clock-synchronization failure, (c) Effect of CPU & network interference on QoT, and (d) Inter-cluster QoT estimation. Note the different y-axis on each plot 113
5.6 Validating the accuracy of the QoT bounds for NTP 114
5.7 Quartz QoT estimation: (a) PTP and (b) Huygens 116
5.8 Quartz PTP: Adaptive clock-synchronization with QoT requirement (a) 10μs, (b) 5μs, (c) QoT bounds on clock-synchronization failure, and (d) Effect of CPU & network interference on QoT. Note the different y-axis on each plot . . 117

6.1 The taskset $\tau_1 = (1, 10)$, $\tau_2 = (4, 23)$, $\tau_3 = (3, 36)$ being scheduled with ES-RHS on the top, and ES-RHS+ at the bottom. For both cases, $C_{sleep} = 5, T_{sleep} = 10$. 127
6.2 Approximation ratio for ES-RMS and WFD as a function of $U_{SleepMin}$ 139
6.3 (a) Percentage of tasksets schedulable with respect to taskset utilization, and (b) Utilization of the forced sleep and total deep sleep with respect to taskset utilization, in the uniprocessor context 144
6.4 (a) Percentage of tasksets schedulable with respect to the minimum supported sleep duration, $C_{SleepMin}$, and (b) Utilization of the forced sleep and total deep sleep with respect to the minimum supported sleep duration, $C_{SleepMin}$, in the uniprocessor context 145
6.5 (a) Percentage of tasksets schedulable with respect to taskset utilization, and (b) Utilization of the synchronous forced-sleep execution versus taskset utilization, in the multi-processor *SyncSleep* scheduling context ($m = 4$) 146
6.6 (a) Percentage of tasksets schedulable with respect to taskset utilization, and (b) Utilization of the synchronous forced-sleep execution versus taskset utilization, in the multi-processor *SyncSleep* scheduling context ($m = 8$) 147
6.7 Guaranteed deep-sleep utilization versus taskset utilization, in the multi-processor *IndSleep* scheduling context ($m = 4$) 148

7.1 *SyncSleep* Scheduling for a quad-core system with cores M_i. 161
7.2 *IndSleep* Scheduling with uniform sleep periods for a quad-core system with cores M_i. 166
7.3 Uniprocessor Results (a) Utilization of the ES-task w.r.t taskset utilization, and (b) % of task sets schedulable w.r.t taskset utilization 170
7.4 Uniprocessor Results (a) Worst-case maximum temperature w.r.t taskset utilization, and (b) Worst-case maximum temperature w.r.t $C_{SleepMin}$ 170

7.5 SyncSleep Results (a) % of task sets schedulable w.r.t taskset utilization, for multi-core *SyncSleep* scheduling ($m = 4$), and (b) Worst-case maximum temperature w.r.t taskset utilization, for multi-core *SyncSleep* scheduling ($m = 4$) . 171
7.6 SyncSleep Results (a) Synchronized ES-task utilization w.r.t taskset utilization, for multi-core *SyncSleep* scheduling ($m = 4$), and (b) Worst-case maximum temperature w.r.t taskset utilization, for multi-core *SyncSleep* scheduling ($m = 8$)172
7.7 IndSleep Results (a) % of task sets schedulable w.r.t taskset utilization, for multi-core *IndSleep* scheduling ($m = 4$), and (b) Worst-case maximum temperature w.r.t taskset utilization, for multi-core *IndSleep* scheduling ($m = 4$) . . 173
7.8 Dynamic simulations (a) maximum temperature w.r.t taskset utilization ($m = 4$), and (b) average power w.r.t taskset utilization ($m = 4$) 174

8.1 (a) Energy and (b) Accelerator Frequency vs CPU Frequency 198
8.2 Energy as a function of (a) CPU Utilization, (b) Accelerator Utilization 205
8.3 Energy as a function of (a) % of tasks using the accelerator (b) power-model parameters 206
8.4 CPU and Accelerator frequency as a function of (a) CPU Utilization, (b) Accelerator Utilization, and (c) Percentage of tasks using the accelerator 207
8.5 WFD vs Sync-Aware WFD (SA-WFD) (a) schedulability comparisons, (b) Energy vs CPU utilization, and (c) Energy vs Accelerator utilization 208
8.6 Multicore Independent Frequency normalized energy vs CPU utilization (a) independent frequency vs uniform frequency, and (b) WFD vs SA-WFD 209

9.1 Concurrency-Induced Serialization on the CPU 219
9.2 Lack of a Critical Instant for Concurrent GPUs 221
9.3 Example framing the blocking calculation as a bin-packing problem 223
9.4 Liquefaction Example 226
9.5 Wavefront example 231
9.6 Accelerator-Partitioning Example 241

9.7 Schedulability vs CPU Utilization: (a) unicore: concurrent analyses, (b) unicore, (c) 4-core and, (d) 8-core concurrent vs traditional analysis. 245
9.8 Schedulability vs Taskset Accelerator Parameters: (a) accelerator utilization, (b) maximum accelerator fractional requirement of task requests, (c) maximum number of per-task accelerator segments and, (d) percentage of tasks using the accelerator. 248
9.9 Uniprocessor + Partitioned-Accelerator Schedulability: (a) accelerator utilization, and (b) maximum accelerator fractional requirement of task requests . . . 249
9.10 Multicore + Partitioned-Accelerator Schedulability: (a) accelerator utilization, and (b) maximum accelerator fractional requirement of task requests 250

Chapter 1

Introduction

> The only reason for time is so that everything doesn't happen at once.
>
> *Albert Einstein*

1.1 Motivation

Coordination is key to the successful operation of a distributed cyber-physical system. Distributed coordination occurs at different spatial and temporal scales, ranging from local-scale robotic coordination – occurring at the timescale of hundred microseconds to a few milliseconds, and city-scale connected vehicles coordinating at the granularity of hundreds of microseconds to a few milliseconds, to planetary-scale coordination among GPS satellites – occurring even at the nanosecond timescale. A non-exhaustive list of such coordinated systems includes swarm robotics [5], distributed databases [6], tele-surgery [7], industrial robotics [8], smart grids [9] and connected vehicles [10]. Figure 1.1 highlights the spatio-temporal nature of distributed coordination at different scales.

The common thread binding many of the above-mentioned applications is the need for low-latency decision-making. This is particularly true for *cyber-physical* systems

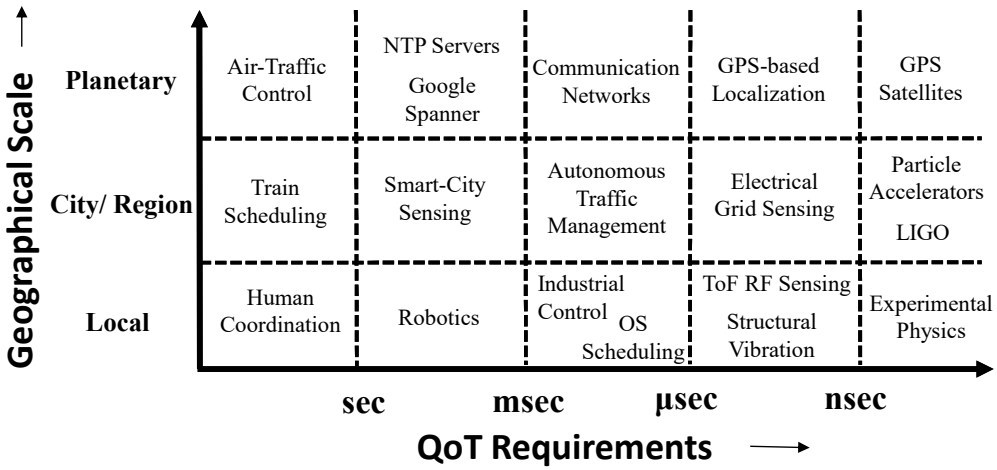


Figure 1.1: The scale of coordination in time and space

(CPS) [11], which involve the cyber components of computation and networking, interacting with and controlling elements in the physical world [11]. In these systems, the nature of coordination is usually dependent on the analysis of *sensed* data by intelligent *computational* entities, which in real-time decide a course of coordinated *action/actuation* at distributed endpoints. The data-intensive and low-latency nature of decision-making makes the cloud in tandem with edge cloudlets and embedded endpoints well-suited for hosting such applications. While recent work [12] [13] [14] [15] has focused on the need for edge computing to reduce the latency of computation, the need for a distributed coordination primitive has not received much attention.

Time is one such construct which plays an important role in enabling coordination among distributed entities [16]. This is especially true for cyber-physical systems (CPS) which need to interact with the real world. A shared notion of time, by means of synchronized clocks, enables [17]:

1. events to be ordered at distributed scale, and

2. coordinated actuation to be scheduled at/by specific time instants.

Therefore, maintaining a shared notion of time is critical to the performance and reliable operation of many large-scale distributed systems.

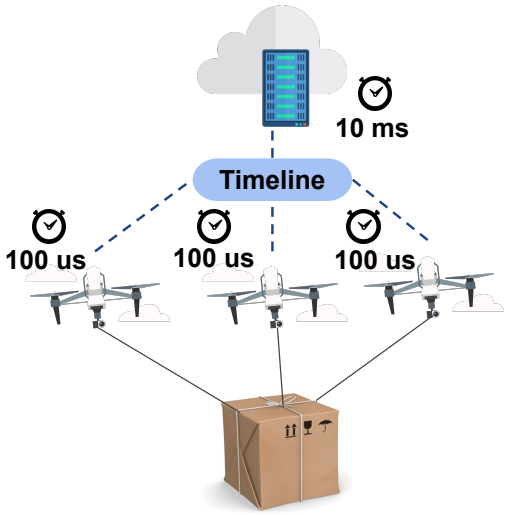


Figure 1.2: *DronePorter*: Time-based Drone coordination

To illustrate the importance of a shared notion of time, we consider a fleet of n drones (as shown in Figure 1.2) transporting an object Ω, too large to be carried by a single drone. We call this hypothetical application DronePorter. To successfully transport Ω, the drones need to follow a coordinated flight-plan such that (i) the object is not damaged or destabilized, and (ii) the drones do not collide with each other or obstacles in the environment. One way to accomplish this is by having a master entity, which can be one of the drones, send out timestamped flight-plans with way-points to each of the drones, such that each drone tries to reach a given way-point at the specified time.

To coordinate successfully, the clock on each drone needs to be synchronized such that the accuracy is within some specified limits. This accuracy (or uncertainty) specification can depend on multiple factors, ranging from the velocity and size of the drones, to the other uncertainties in the environment. For example, to meet a particular velocity, while maintaining safety, having a tighter clock-synchronization accuracy can be used to compensate for higher localization uncertainties or higher environmental uncertainties [18]. Additionally, as shown in Figure 1.2, we can also have an edge/cloud controller which provides (i) high-level objectives/guidance to the fleet of coordinating drones, and (ii) fleet-management capabilities. One can also envision that this edge/cloud-controller can be responsible for multiple fleets of drones.

While a shared notion of time is key for geo-distributed coordination, it also helps characterize a system's end-to-end latency requirements, necessary for enabling real-time decision making. These low-latency requirements are characterized by per-component deadlines, which need to be met to satisfy the safety and/or *end-to-end* performance specifications of the system. Hence, it also becomes necessary to effectively *schedule* multiple application components at each compute *node*, using analyzable real-time scheduling techniques such that they meet all deadlines.

Let us re-visit the DronePorter application. Each drone relies on processing multiple sensor-driven data streams, such as cameras and LIDARs, to perceive its surroundings. To operate safely, these data streams need to be analyzed in real-time, in order to decide an appropriate course of action before a deadline. Techniques ranging from signal processing to machine learning, which are both computationally intensive and highly parallelizable, are often used in the decision-making process. These observations also apply to multiple cyber-physical application domains including, but not limited to, autonomous vehicles [19], augmented reality [20] and robotics [8]. Therefore, the scheduling techniques utilized in such cyber-physical systems need to take into account,

- the computational requirements of each application, and their execution patterns,
- the use of heterogeneous platforms with multi-core processors coupled with one or more concurrent hardware accelerators such as General-Purpose Graphics Processing Units (GP-GPUs), Digital Signal Processors (DSPs), and Application-Specific ICs (ASICs) to meet the increasing computational requirements [21] [22] [23], and
- physical limitations such as energy budgets and thermal constraints, because CPS are often deployed in resource-constrained or mobile settings.

Therefore, while a shared notion of time enables coordination at the *distributed* scope, to coordinate effectively it is also necessary to simultaneously schedule multiple application components at the scope of each *node*, such that all deadlines are met, while

ensuring that the resource/physical constraints of the system are satisfied. Hence, in this dissertation we introduce:

1. abstractions, system-design methodologies and frameworks that enable time-based coordination in geo-distributed cyber-physical systems, and
2. energy-, thermal- and resource-efficient analyzable real-time scheduling techniques for applications deployed on platforms utilizing both multi-core processors and hardware accelerators.

The thesis supported by this dissertation is as follows:

> **Thesis Statement:** *Time-sensitive cyber-physical applications can effectively coordinate multiple geo-distributed components deployed across the cloud, edge cloudlets and resource-constrained embedded platforms, by: (i) utilizing our time-as-a-service abstraction, which provides a shared, precise and adaptive notion of time based on application-defined quality metrics, and (ii) low-latency computations made predictable and energy-efficient by adopting analyzable real-time scheduling techniques.*

The remainder of this chapter provides an overview of this dissertation. We first describe the scope of this dissertation by providing an overview of the problem statements we consider. Subsequently, we briefly describe the key contributions of this dissertation. Lastly, we state the organization of the rest of this dissertation.

1.2 Scope of the Thesis

We now describe the key problem spaces we address in this dissertation, and briefly outline the direction of the solutions we propose.

1.2.1 Time-based Distributed Cyber-Physical Coordination

Consider a distributed cyber-physical application with components performing sensing, actuation and computation deployed across multiple geo-distributed nodes. Reliable planetary-scale coordination among these application components requires a shared and precise notion of time [4]. Clock synchronization is a mature field and technologies such as the Global Positioning System (GPS), Network Time Protocol (NTP) [1], and Precision Time Protocol (PTP) [2] have made it possible to provide distributed systems with a reliable and accurate shared notion of time. However, these technologies are best-effort and/or agnostic to application-specific requirements. Additionally, clock synchronization is not perfect, and there is always some uncertainty in a node's estimate of the shared notion of time. This timing uncertainty is introduced by a variety of factors including, but not limited to, networking delays [1], timestamping errors, and operating system and virtualization-induced latency and jitter [24] [25]. If this timing uncertainty exceeds an application's specifications, it can affect the quality and reliability of coordination [17]. The level of uncertainty acceptable to an application often depends on the time granularity at which coordination occurs, as well as the coordination policy itself [17].

As time is fundamental to a range of applications, it needs to be exposed *as a service* to applications. Therefore, we propose and define *Time-as-a-Service* (TaaS) as,

> "the ability to provide an application-specific clock, which tracks a time reference, such that the timing uncertainty is within application-specified requirements."

Time exposed as a *first-class entity* to applications addresses these issues effectively. This can be done by: (i) allowing applications to specify their timing requirements in terms of accuracy and resolution, and (ii) feeding back the delivered timing uncertainty back to the application. Allowing distributed application components the ability to specify their uncertainty tolerances enables the underlying system to orchestrate the

infrastructure to meet them, thus providing "time-as-a-service". Furthermore, exposing the delivered uncertainty to applications allows them to be fault-tolerant and adaptive in the event where the delivered uncertainty exceeds specified limits. Thus, fault-tolerant time-based coordination becomes enabled by using our notion of *Quality of Time* (QoT) [4], which represents,

> "the *end-to-end* uncertainty bounds corresponding to a timestamp, with respect to a clock reference."

From an application perspective, if these bounds exceed an acceptable limit, the application can enter a graceful degradation mode, and thus be fault-tolerant during clock-synchronization failure. For instance, let us re-visit the DronePorter application described in Figure 1.2. If the QoT deviates beyond the specified requirements, the drones can be notified, and can adapt by gracefully coming to a safe halt on the ground.

Therefore, we propose to leverage Quality of Time (QoT) to develop software abstractions necessary to expose "Time-as-a-Service" to applications, along with a QoT-based application programming interface (API) to enable fault-tolerant time-based coordination in cyber-physical systems.

Modern distributed cyber-physical applications are inherently complex, and consist of multiple interacting components. Thus, deploying these components and managing their life-cycles are complex endeavors. Additionally, while some components may be deployed on bare-metal embedded devices, many of these components will also be deployed in the cloud or at the edge in conjunction with other applications. In such scenarios, the use of virtualization technologies like virtual machines [26] and containerization [27] simplifies the deployment and life-cycle management of distributed applications. Therefore, we focus on the challenges associated with delivering Time-as-a-Service, at both the node[1] scope [4] [24] as well as the distributed scope [28].

[1]We use the terminology "node" to define an independent physical or virtual computing platform.

1.2.2 Fixed-Priority Real-Time Scheduling

Scheduling application components within the node scope at/by the right time instant is key to ensuring effective coordination in cyber-physical systems. As mentioned in Section 1.1, given that most cyber-physical systems also have safety and performance requirements, we need to ensure that each application component needs to complete before its respective deadline. Therefore, we utilize analyzable real-time scheduling techniques to provably check if all deadlines can be met. This subsection provides an overview of the real-time scheduling terminology we use in this dissertation, as well as some of the assumptions we make with respect to scheduling policy and computational platforms.

Terminology: In the terminology of real-time scheduling each *application component* deployed on a node is referred to as a *task* [29], and a collection of tasks deployed on a node is referred to as a *taskset*. These cyber-physical tasks typically perform recurrent operations which are either triggered at specific time instants (*time-triggered*) or by the occurrence of an event (*event-triggered*). Hence, we utilize the *sporadic task model* to model CPS applications. Under, the sporadic task model, each task repeatedly releases a workload, called a *job*, with a minimum inter-arrival time between two subsequent job arrivals of a single task. The *response time* of a task is defined as the time duration from when a job of the task is released till the time it completes. Therefore, for a task to always be schedulable, its *worst-case* response time must always be less than or equal to its deadline. Given a real-time scheduling policy, if we can show that all tasks τ_i in a taskset Γ *always* meet their respective deadlines, then the taskset is said to be *schedulable*.

Scheduling Policy: In this dissertation, we focus on fixed-priority real-time scheduling [29], as it is analyzable and widely supported in many commercial [30] [31] and open-source operating systems [32] [33]. In the realm of multi-core real-time scheduling, there are two approaches to scheduling tasks on multiple processing cores: (i) partitioned scheduling, and (ii) and global scheduling. Partitioned scheduling statically

assigns each task to a core and tasks always execute on the assigned core. Therefore, in the partitioned-scheduling context, finding an optimal task allocation can be modeled as a bin-packing problem. On the other hand, global scheduling allows tasks to migrate between cores at runtime. In this dissertation, we focus on partitioned multi-core scheduling as it yields more predictable execution behavior at run time, delivers much better worst-case performance, and unlike global scheduling does not suffer from scheduling anomalies [34].

Platform: Modern CPS rely on analyzing computationally-intensive data streams in real-time to perform decision making. Therefore, it is common to find CPS platforms containing multi-core processors coupled with one or more hardware accelerators such as GPUs and DSPs. In this dissertation, we consider platforms with homogeneous multi-core processors, where each CPU core has identical characteristics. We also consider hardware accelerators which do not support preemption. This assumption is in line with the fact that most commercially-available hardware accelerators, including GPUs, do not support preemption. These hardware accelerators may support *concurrent* execution of multiple tasks [35], or may only allow mutually-exclusive access to a single task. Additionally, we also consider the possibility of modern hardware accelerators supporting software partitioning of computational resources [36] [37].

1.2.3 Energy and Thermal-Aware Real-Time Scheduling

Cyber-physical systems often have components deployed in mobile and/or resource-constrained settings [5] [19]. Therefore, we also need to take into account the physical constraints of the system while meeting all task deadlines. In particular, we focus on energy and thermal constraints, where reducing the energy consumption of a battery-powered system can lead to a significant increase in operating lifetime [5], while decreasing the operating temperature of the system can lead to increased reliability and prevent thermal failure [38] [39].

Advancements in semiconductor technology have enabled compute-intensive cyber-physical applications by increasing the number of transistors available to system designers. However, the side effects of rising transistor density include increased power and heat dissipation [40]. Therefore, energy savings and system temperature are intricately tied together. Modern processors are equipped with energy-management features such as Dynamic Voltage and Frequency Scaling (DVFS) [41], and the use of low-power sleep states [42]. DVFS enables the processor to change its operating frequency and voltage, thereby reducing *dynamic switching power*, while low-power sleep states use power gating and/or clock gating [43] to reduce *static leakage power* dissipation when the processor is idle. As transistor geometries get smaller, the dominance of static power as a contributor to total power consumption is only expected to increase [44]. Since static power is also directly dependent on the operating temperature, scheduling techniques will increasingly need to take advantage of processor sleep states.

Most modern multi-core processors support a number of low-power states called C-states [45]. However, there is a minimum *round-trip* time associated with each of these low-power sleep states [42]. This round-trip time is longer for moving to and from lower-power states due to the overhead required for the main oscillator to startup and stabilize [42]. From a real-time systems scheduling perspective, it is critical that all tasks τ_i in a given taskset Γ meet their deadlines to ensure reliable system operation. Therefore, it is essential that the processor be put into a *correct* sleep state at the *correct* time and for a pre-computed duration, to ensure that all deadlines are met, while the energy consumption of the system is minimized.

In particular, we focus on designing energy-efficient multi-core partitioned fixed-priority real-time scheduling techniques, which utilize the processor's deep-sleep state to reduce static leakage power, and hence save energy. We also focus on analyzing the thermal implications of utilizing these techniques on real-world multi-core platforms.

Hardware accelerators are also commonly found in CPS platforms and consume significant amounts of power [46]. Like multi-core processors, hardware accelerators can

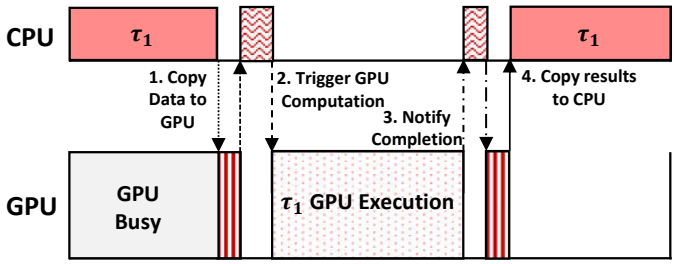


Figure 1.3: Task τ_1 executing using a combination of CPU and GPU resources

also expose power-management interfaces. However, in commercial accelerators like GP-GPUs and DSPs, only P-states are exposed to the user [47] [48]. Thus, in effect, they expose only voltage and frequency-scaling knobs for power management, and the job of reducing static power is done in firmware or hardware. Therefore, we focus on using frequency-scaling-based techniques to reduce the power consumption of systems using hardware accelerators. In particular, we focus on techniques to statically set the processor and accelerator to a pre-computed taskset-specific frequency, such that the aggregate energy consumption is reduced, while ensuring that all deadlines are met. Therefore, as there are no dynamic frequency changes, the unpredictable latency involved in changing the oscillator frequency is avoided, leading to more deterministic operation, which is desirable in real-time systems.

Consider a platform with a multi-core processor coupled with one (or more) hardware accelerator(s). As shown in Figure 1.3, tasks executing on this platform execute using a combination of both CPU and accelerator resources. In this scenario, to meet all task deadlines while reducing energy consumption, we jointly optimize the accelerator and CPU frequencies in order to reduce the energy consumption of the entire system.

1.2.4 Real-Time Scheduling for Concurrent Accelerators

As mentioned in Section 1.1, CPS ranging from autonomous vehicles to industrial robots increasingly rely on techniques such as deep-neural networks for perception and planning. Additionally, it is not uncommon to find multiple such computationally-intensive workloads as part of a single application. Modern accelerators often support concurrent

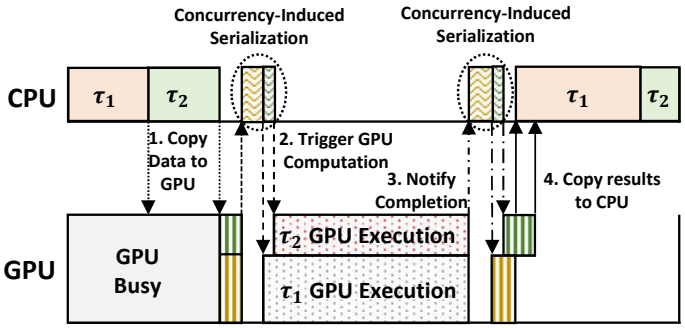


Figure 1.4: Tasks τ_1 and τ_2 executing concurrently on the GPU

execution, and allow requests belonging to different tasks to be *co-scheduled* and execute in parallel. This is especially true for modern GPU architectures such as NVIDIA Fermi and Pascal [49] [50]. For example, the NVIDIA Xavier [21] has 512 cores which can be utilized by concurrent kernels. Such platforms often provide built-in schedulers which aim to maximize concurrency, but do not take into account task deadlines.

From a real-time systems perspective, for a set of tasks to be schedulable, it is imperative that all deadlines be met. Therefore, multiple analytical frameworks [51] [52] [53] [54] [55] have been proposed to analyze the schedulability of tasksets which utilize hardware accelerators such as GP-GPUs. To the best of our knowledge, most known analysis techniques treat the accelerator as a mutually-exclusive resource which at any point of time can only be accessed by a single task. This leads to additional schedulability pessimism for accelerators supporting concurrent execution. Therefore, we focus on developing schedulability-analysis techniques for real-time tasksets utilizing hardware accelerators which support concurrent execution. Figure 1.4 shows an example sequence of two tasks executing concurrently on a GPU.

In terms of scheduling policies, we focus on work-conserving fixed-priority scheduling and non-work-conserving First-in-First-out (FIFO) scheduling. Traditionally, co-scheduling task requests concurrently on hardware accelerators like GPUs has considered the *global* scheduling paradigm. In this paradigm, task requests are ordered in a single queue [56] and dispatched to be scheduled on any part of the global resource. However, recent GPU architectures such as NVIDIA Volta [36] coupled with software-

partitioning techniques [37] have enabled GPUs to be partitioned into multiple fractional components. Therefore, we even propose techniques for partitioning a hardware accelerator, to increase execution predictability.

1.3 Contributions

The primary contribution of this thesis is the development of novel distributed software abstractions and frameworks, which in conjunction with node-level analytical real-time scheduling techniques, enable resource-efficient and time-aware geo-distributed coordination in cyber-physical systems.

We now briefly describe each of the individual contributions which make up this dissertation.

1.3.1 Making Time *Prime* in Cyber-Physical Systems

We now briefly summarize our contributions which enable time-based geo-distributed coordination in CPS. Detailed descriptions of these contributions can be found in Chapters 3, 4 and 5.

- **Timelines, Quality of Time (QoT) and the QoT Stack [17] [4]:** Adopting a *holistic* notion of Quality of Time (QoT) that captures clock metrics such as resolution, accuracy, and stability, we propose an architecture in which the local perception of time is a controllable operating system primitive with observable uncertainty, and where clock synchronization balances applications' timing demands with system resources such as energy and bandwidth. Our architecture features an expressive application programming interface that is centered around the notion of a *timeline* – a virtual temporal coordinate frame that is defined by an application to provide its distributed components with a shared sense of time, with a desired *accuracy* and *resolution* – that enables developers to easily write applications whose activi-

ties are choreographed across time and space. The *timeline* abstraction on the one hand allows applications to adapt to changes in uncertainty in system time, and on the other hand enables the OS to efficiently manage clocks and synchronization protocols. Leveraging open-source hardware and software components, we prototype an implementation for Linux called the *QoT Stack*, and present results from its evaluation on a standard embedded-computing platform.

- **Bringing QoT to Virtual Machines [24]:** Given that most public clouds and edge cloudlets provide multi-tenancy using *virtualized* units of computing, we aim to introduce the notion of QoT to virtual machines. The use of virtual machines entails the use of a hypervisor, which adds additional timing uncertainty due to relatively higher jitter in clock-read and timer-interrupt latencies. Hence, the use of virtualization presents a challenge in terms of observing and guaranteeing the QoT delivered to an application. To meet these challenges, we present the *QuartzV* extension to the QoT Stack, to make virtual machines QoT-aware. We utilize the open-source QEMU-KVM [57] hypervisor, and illustrate the *para-virtual* design choices that are key for delivering near-native levels of *timing* performance in virtual machines. We also demonstrate the utility of QuartzV by using it in the development of an industrial-automation application. Our experimental evaluations also show the efficacy of QuartzV with respect to the native and fully-virtualized cases.

- **Time-as-a-Service for Geo-distributed Coordination [18] [28]:** The emergence of edge computing, specifically to facilitate low-latency decision-making, is leveraging the trend where multiple cyber-physical and software applications with different *timing* requirements will coexist in both the cloud and at the edge. To enable such fault-tolerant *time-based* coordinated applications running on multi-tenant geo-scale infrastructure, we introduce the *Quartz* framework, which exposes *Time-as-a-Service*. Quartz allows geo-distributed application components to each specify its timing requirements, while it *autonomously* orchestrates the underlying

infrastructure to meet them. Centered around a shared *virtualized* notion of time, based on the *timeline* abstraction, Quartz provides an API which makes it easy to develop time-based geo-distributed applications. Using this API, Quartz feeds back the timing uncertainty, i.e., the delivered *Quality of Time (QoT)* back to each application, enabling it to be fault-tolerant in the face of clock-synchronization failure. Quartz is designed for containerized applications, features a distributed architecture and is implemented using containerized micro-services. Our experimental evaluations on real-world embedded, edge and cloud platforms highlight the performance and scalability of our architecture.

1.3.2 Analytical Techniques for Energy-Aware Real-Time Scheduling

We now briefly describe the energy-aware real-time scheduling policies which make up this dissertation, along with their corresponding schedulability-analysis techniques. Detailed descriptions of these contributions can be found in Chapters 6, 7 and 8.

- **Energy-Saving Multi-core Real-Time Sleep Schedulers [58]:** Modern processors provide sleep states which minimize leakage power by gating portions of the processor and/or the system clock. We present partitioned fixed-priority scheduling solutions for utilizing these sleep states to efficiently schedule sporadic real-time tasks, and maximize energy savings on multi-core processors. The techniques presented rely on an enhanced version of *Energy-Saving Rate-Harmonized Scheduling* (ES-RHS) [42], and our newly proposed *Energy-Saving Rate-Monotonic Scheduling* (ES-RMS) policy to maximize the time the processor spends in the lowest-power *deep-sleep* state. We collectively call these schedulers *Energy-Saving* schedulers. In some modern multi-core processors, all cores need to transition synchronously into deep sleep. For this class of processors, we present a partitioning technique called Max-SyncSleep which utilizes *a priori* task information, to maximize the synchronous deep-sleep duration across all processing cores. The performance of Max-

SyncSleep is compared to the classical *Worst-Fit Decreasing* load-balancing heuristic. We also illustrate the benefits of using ES-RMS over ES-RHS for this class of processors. For processors which allow cores to individually transition into deep sleep, we prove that, while utilizing ES-RHS on each core, any feasible partition can optimally utilize all of the processor's idle durations to put it into deep sleep. Our experimental evaluations indicate that our proposed techniques can provide significant energy savings and better schedulability.

- **Thermal Implications of Energy-Saving Schedulers [59]:** In many real-time systems, continuous operation can raise processor temperature, potentially leading to system failure, bodily harm to users, or a reduction in the functional lifetime of a system. In this dissertation, we explore the relationship between energy savings and system temperature in the context of fixed-priority *energy-saving* schedulers, which utilize a processor's *deep-sleep* state to save energy. We derive insights from a well-known thermal model, and are able to identify *proactive* design choices which are *independent* of system constants and can be used to reduce processor temperature. Our observations indicate that, while energy savings are key to lower temperatures, not all energy-efficient solutions yield low temperatures. Based on these insights, we propose the SysSleep and ThermoSleep algorithms, which enable a thermally-effective sleep schedule. We also derive a lower bound on the optimal temperature achievable by energy-saving schedulers. Additionally, we discuss partitioning and task-phasing techniques for multi-core processors, which require all cores to synchronously transition into deep sleep, as well as those which support independent deep-sleep transitions. We observe that, while energy optimization is straightforward in some cases, the dependence of temperature on partitioning and task phasing makes temperature minimization non-trivial. Our evaluations show that compared to the existing purely energy-efficient design methodology, our proposed techniques yield lower temperatures (up to ~4°K lower) along with

significant energy savings.

- **Energy-Saving Scheduling for Real-Time Systems with Hardware Accelerators [60]:** In CPS, most tasks execute using a combination of CPU and accelerator resources. Hence, the power of the CPU and the accelerator needs to be managed in *tandem*. To reduce energy consumption, commercially-available accelerators such as GP-GPUs and DSPs expose interfaces to scale their operating voltage and frequency. Hence, we propose the CycleTandem static frequency-scaling technique to *co-optimize* the operating frequencies of both the CPU and the hardware accelerator. Based on practical considerations of real-world platforms, we consider various energy-management scenarios where the accelerator or CPU frequencies may or may not be adjustable, and propose the CycleSolo family of algorithms for such contexts. Furthermore, we also study partitioning techniques to reduce the operating frequency when multi-core processors are used in conjunction with hardware accelerators. Our experimental evaluations indicate that our proposed techniques can yield significant energy savings. We also present a case-study on the NVIDIA TX2 embedded platform to illustrate the energy savings delivered by our proposed techniques, and observe *up to 44.29% lower energy consumption* as compared to the case without energy management.

1.3.3 Analytical Real-Time Scheduling for Concurrent Accelerators

We summarize the fixed-priority schedulability analysis that we propose for concurrent accelerators. A detailed description of this analysis can be found in Chapter 9.

- **Co-Scheduling Real-Time Workloads on Concurrent Hardware Accelerators:** Modern accelerators often support concurrent execution, and allow requests belonging to different tasks to be *co-scheduled* and execute in parallel. However, existing fixed-priority real-time scheduling analyses assume that tasks can access the accelerator only one at a time. This leads to additional schedulability pessimism

for accelerators supporting concurrent execution. In this dissertation, we propose schedulability-analysis techniques for real-time tasksets utilizing hardware accelerators which support concurrent execution. In terms of scheduling policies, we focus on work-conserving fixed-priority scheduling and non-work-conserving FIFO scheduling. We consider *global* scheduling, where the accelerator is treated as a single resource. Our experimental evaluations suggest that our proposed analysis methodologies can yield improved schedulability, up to ~2x more tasksets, over traditional non-concurrent analysis techniques.

- **Partitioning Techniques for Concurrent Hardware Accelerators:** Modern GPU architectures [36] coupled with software-partitioning techniques [37] have enabled GPUs to be partitioned into multiple fractional components. Therefore, we also consider *partitioned*-scheduling techniques, where an accelerator can be partitioned into discrete units, and the accelerator requests in the taskset can be allocated to these partitions. In particular, we propose a novel worst-fit decreasing-based heuristic to create accelerator partitions, and allocate requests to them. Our experiments indicate that, as compared to the global scheduling-based approach, our proposed partitioning techniques offer improved schedulability.

1.4 Organization

The rest of this dissertation is organized as follows. Chapter 2 reviews the relevant prior work. Chapters 3, 4 and 5 describe abstractions and frameworks which enable time-based coordination in geo-distributed CPS. Chapters 6 and 7 introduce the family of energy-saving sleep schedulers, and provide an analytical understanding of the energy savings they provide along with their thermal implications. Chapter 8 introduces the CycleSolo and CycleTandem algorithms to reduce the energy consumption of real-time systems using hardware accelerators. Chapter 9 provides techniques and schedulabil-

ity analysis for real-time systems using concurrent hardware accelerators. Chapter 10 concludes this dissertation and provides possible future research directions.

Chapter 2

Related Work

This chapter presents the relevant prior work on topics in the scope of this dissertation. In particular, we cover relevant clock-synchronization techniques and time-based system-design methodologies, energy- and thermal-aware real-time scheduling, and real-time resource management for hardware accelerators.

2.1 Time-based Distributed Coordination

The utility of a shared notion of time in distributed systems has been well-studied in prior work. In [16], the benefits of using synchronized clocks in distributed systems was analyzed. The author concluded that synchronized clocks can improve performance by replacing communication with local computation [16], i.e., by knowing the current time, some properties of the system can be inferred. Therefore, this section provides an overview of clock-synchronization techniques and time-based distributed architectures.

2.1.1 Clock Synchronization

Clock synchronization is a mature field of study, and comprehensive software solutions with accompanying hardware are readily available. Examples of hardware include GPS and atomic clocks, switches that calculate residency delay for routed packets, and

network adapters that precisely timestamp incoming and outgoing packets. Such infrastructure is already in use by many back-end systems. For example, cellular telephone backhaul networks use bespoke hardware for synchronizing transmissions to maximize channel utilization. On the other hand, software protocols like the Network Time Protocol (NTP) [1], and Precision Time Protocol (PTP) [2] play a key role in achieving a reliable and accurate shared notion of time.

The common thread underlying most clock-synchronization protocols involves passing timestamped messages between multiple nodes to estimate the round-trip packet delay, and subsequently the clock offset between them. Most often, calculating the offset Θ between a pair of nodes involves taking the difference of the transmission time of a clock-synchronization packet on the sender node t_{tx}, and the reception time of the packet on the receiver node t_{rx}, and subtracting the estimated packet delay between the nodes Δ. Therefore, the offset can be calculated by:

$$\Theta = t_{rx} - t_{tx} - \Delta \tag{2.1}$$

Multiple such measurements are typically used to compensate for timing uncertainty. This is because, as mentioned in Section 1.2.1, this timing uncertainty is introduced by a variety of factors including, but not limited to, networking delays [1], timestamping errors, and operating system and virtualization-induced latency and jitter [24] [25].

It is currently possible to synchronize to an accuracy in the order of milliseconds with the Network Time Protocol (NTP) [1] over Ethernet, or microseconds with the Precision Time Protocol (PTP) [2] and compliant hardware. More specialized projects, such as WhiteRabbit [61], attain sub-nanosecond error – enough to measure the distance light travels in a second with millimeter accuracy – by compensating for cable delay asymmetry and using Synchronous Ethernet to frequency-lock devices.

Most recently, a number of protocols [3] [62] have been proposed to achieve nanosecond-accuracy clock synchronization in data-centers. Notable among these is the Huygens [3] protocol which uses a peer-to-peer probing mesh along with Support

Vector Machines (SVM) to compute clock offsets between nodes. Based on these offset values, the network effect [3] is used to minimize the clock-synchronization error. The network effect utilizes the fact that clock synchronization is reflexive and transitive, and hence the sum of the clock-synchronization error over a loop should equal zero [3].

In the wireless sensor-networking literature clock synchronization has been approached in a different way. Rather than considering the objective as synchronizing devices to some universal time reference, all that matters is that peer devices – which may be multiple hops away from each other – share a common sense of time, with an emphasis on channel-utilization efficiency. For the case where the root time is maintained across all nodes in the network, Flooding Time Synchronization Protocol (FTSP) [63] is state of the art. It allows only one-way reference broadcasts as opposed to Timing-sync Protocol for Sensor Networks (TPSN) [64] and Reference Broadcast Synchronization (RBS) [65], which use two-way message exchange between nodes, thus effectively reducing the network traffic. All of these protocols implicitly assume that the distance between devices is sufficiently small enough (tens of meters or less) so that propagation delay can be ignored. Recently, Glossy [66] and PulseSync [67] have emerged and improved on the multi-hop accuracy of FTSP by flooding pulses at high speed throughout the network.

However, all the above-mentioned protocols are best-effort and do not consider application-specific QoT requirements. Therefore, most systems using these protocols end up being over-engineered to meet the needs of pre-determined applications. Hence, there is a need for an application-level framework which can respond to application timing demands, while making it easy to develop time-based distributed applications.

Also relevant but complementary to our work is research on analytical modeling of clock uncertainties [68], and methods to compensate for them via approaches such as Kalman filtering [69]. The work in [68] models the clock by applying a Kalman filter to track the clock offset and skew, and compensates for the uncertainty by adjusting the synchronization interval. Seong et al. [70] compensates for the quantization error in

timestamping using a feed forward filter preceding a PI controller. Xu et al. [69] uses a Kalman filter based proportional-integral (PI) clock servo to correct for the quantization error and clock offset. These kind of techniques only discuss few sources of clock uncertainties and try to compensate for them.

In this dissertation, we focus on proposing system abstractions and frameworks, which build on existing clock-synchronization technologies to provide "Time-as-a-Service" to geo-distributed cyber-physical applications.

2.1.2 Time-based System Design

In the context of model-based design, PTIDES [71] is a hardware-software co-design framework to model, design and deploy time-critical embedded applications, using a shared notion of time. For safety-critical systems in the automotive and aerospace domains, the Time-Triggered Architecture (TTA) [72] provides a deterministic way to deploy systems using a shared clock.

The Time-Triggered Architecture (TTA) [72] addresses issues in safety-critical real-time systems by establishing a global time-base to specify interaction between nodes, whereas, an event-triggered architectures like PTIDES [71] maps model time to real time, only when systems react or act to the physical world, e.g, sensors and actuators. PtidyOS [73] is a micro kernel for PTIDES that generates target specific code for the PTIDES model and runs on bare metal. However, these frameworks do not consider the utility of the knowledge of timing uncertainty, and often rely on best-effort clock synchronization and/or system modeling in order to achieve a correct outcome. Additionally, both PTIDES and TTA are designed for the embedded domain, and cannot scale to geo-distributed cyber-physical applications which run in distributed heterogeneous environments including the cloud and the edge.

There has also been some work on distributed-programming idioms that support time as a first-class citizen. Examples include Stampede [74] which uses application-

specified "virtual time" as the basis for enabling temporal causality in distributed applications, Stampede-RT [75] which builds on Stampede and allows distributed applications to timestamp events with real-time tags, and Persistent Temporal Streams [76] which unifies in-memory and stable storage temporal events of a given activity. However, while these systems focus on programming abstractions for ordering distributed events, they do not consider the utility of exposing timing uncertainty to applications, which is key to providing time-as-a-service.

Google's geo-distributed Spanner database utilizes synchronized clocks with the uncertainty information to achieve global-scale consistency [6]. However, Spanner is a closed system, is not adaptive, and relies on dedicated infrastructure. Additionally, the TrueTime API [6] is tailored only to database transactions and does not treat the notion of QoT as an application-specified requirement. The same is true for the POSIX API [77] available in many modern OS.

2.2 Energy-Aware Multi-Core Real-Time Scheduling

Energy savings and system temperature are intricately tied together. Modern processors are equipped with energy-management features such as Dynamic Voltage and Frequency Scaling (DVFS) [41], and the use of low-power sleep states [42]. DVFS enables the processor to change its operating frequency and voltage, thereby reducing *dynamic switching power*, while low-power sleep states use power gating and/or clock gating [43] to reduce *static leakage power* dissipation when the processor is idle. As transistor geometries get smaller, the dominance of static power as a contributor to total power consumption is only expected to increase [44]. Since static power is also directly dependent on operating temperature, scheduling techniques will increasingly need to take advantage of processor sleep states.

2.2.1 Energy-Aware Real-Time Scheduling

In the domain of real-time systems, the use of frequency scaling-based energy saving scheduling techniques has been well-studied. In the context of fixed-priority scheduling, Saewong et al. [78] proposed the SysClock algorithm to analytically determine the energy-optimal frequency at which the processor must run, so that a taskset meets all its deadlines. In the same work, dynamic frequency scaling-based scheduling techniques named PM-Clock and DynamicPM-Clock [78] were also proposed. In [79], Arvind et al. proposed the Static Frequency Assignment Algorithm (SFAA) to partition tasks on multi-core processors which have a single frequency domain. SFAA extends the SysClock framework to the multi-core context and aims to minimize the operating frequency across all cores, so as to minimize the energy consumption. In [80] [81] [82], DVFS-based scheduling techniques for multi-core processors can be found, where each core has its own voltage and frequency domain.

At technology nodes smaller than 65nm [83], static leakage power already dominates the total power consumption of CMOS-based VLSI circuits. For general-purpose computing workloads, Le Sueur et al. [45] compared the energy efficiency of DVFS and sleep state-based power-management techniques. Their work experimentally analyzes the trade-offs between *slow down* (DVFS) and *race-to-halt* (sleep) for a range of computing platforms such as the desktop-class Intel i7 870 and the low-power Intel Atom Z550. The result of the analysis concluded that using C-state based techniques offer improved energy efficiency with a small impact on performance [45].

In the context of real-time systems, *a priori* task execution information can be utilized to schedule tasks efficiently so as to maximize the time spent in low-power states. Scheduling techniques that cluster task executions to save energy have been proposed in [84] [85] [86] [87] [88]. In [85] [87] [88], dynamic-priority EDF scheduling is used. [86] uses procrastination of tasks to determine the instances at which the processor can be shutdown. [84] uses a fixed-priority scheduling based approach, which relies on online

simulation to estimate the duration for which a task can procrastinate, thus significantly increasing the scheduler run-time complexity. For tasks with given release times and deadlines, [89] and [90] present dynamic priority-based scheduling techniques to maximize the common idle time on multi-core processors. In [42], Rowe et al. proposed and analyzed the benefits of using ES-RHS in uniprocessor systems. ES-RHS is a simple, easy-to-implement approach based on a notion of *harmonization*, that aggregates all the processor idle durations together. This allows the processor to be put into deep sleep for all idle durations, thus enabling optimal energy savings for processors which lack frequency-scaling capabilities [42].

In conjunction with reservation-based real-time operating systems (such as Linux/RK [91]), ES-RHS presents an effective approach for energy management in uniprocessor real-time systems, and is particularly useful for processors which have only one sleep state [92]. The work by Rowe et al. [42] also offers brief guidelines for the use of ES-RHS in the multi-core context. We build on ES-RHS to propose *deep-sleep*-based energy-saving techniques for real-time systems using multi-core processors.

2.2.2 Thermal-Aware Real-Time Scheduling

Thermal Management can be done reactively at runtime [93] [94] [95] [96] or proactively at design time [97] [98] [99] [100] [101] [102] [103]. In the scope of reactive techniques, Fu. et al. [94] proposed a control-theoretic algorithm to meet the desired temperature requirement on a multi-core processor, subject to timing constraints. Yun et al. [96] used a machine-learning technique (SVM) to predict the temperature profile of a multiprocessor system. Based on the predicted value, a dynamic temperature-management scheme is used. In [93], Chandarli et al. proposed an optimal reactive scheduler for fixed-priority uniprocessor sleep scheduling along with an associated response-time based analysis framework. However, reactive schedulers require temperature sensors, which may not always be present in real platforms.

In the scope of proactive techniques, [101] describes a real-time scheduling algorithm for uniprocessors, based on a thermal model approximated by Fourier's Law. The algorithm derives a speed schedule by minimizing temperature under both timing and thermal constraints. In [102], an assignment and scheduling technique for an MPSoC was proposed, which utilizes a mixed-integer linear program solver to optimize the peak temperature. In [99], an optimal speed schedule is derived for a multi-core platform, based on a thermal model given at design time. In [103], Masud et al. proposed the use of a thermal-aware periodic resource to minimize peak temperature, in the context of uniprocessor Earliest Deadline First (EDF) scheduling. The processor slack is utilized to put the processor into a sleep state.

Most of the pieces of earlier work [94] [95] [96] [99] [97] [98] have focused on the use of DVFS to optimize the processor temperature. However, the dominance of static power makes it necessary to investigate techniques which utilize sleep states. Additionally, many low-powered devices often lack DVFS, but support sleep states [42]. The work in [93] and [103] propose thermal-aware techniques which utilize processor sleep states. However, [103] assumes dynamic-priority EDF scheduling. On the other hand, [93] presents a reactive framework for uniprocessor fixed-priority scheduling. To the best of our knowledge, no thermal-analysis framework for proactive fixed-priority sleep scheduling exists in the literature.

In this dissertation, we analyze the thermal implications of fixed-priority energy-saving schedulers which periodically utilize the processor's deep-sleep state, in light of their energy-saving properties. Based on a well-known thermal model, we derive practical insights and algorithms. Our proposed techniques focus on minimizing the maximum temperature, rather than optimizing to meet a set of thermal constraints.

2.3 Real-Time Scheduling for Hardware Accelerators

In this section, we first summarize the techniques proposed in literature to arbitrate access to hardware accelerators like GP-GPUs. Subsequently, we discuss energy-saving techniques for hardware accelerators.

2.3.1 Accelerator as a Mutually-Exclusive Resource

Most commercially-available accelerators like GP-GPUs and DSPs do not typically support preemption. The large number of registers in these accelerators makes context switching an expensive proposition. Therefore, prior work [53] [52] has focused on modeling accelerator access as a critical section arbitrated by a global lock. In particular, the work in [54] [104] models GPUs as mutually-exclusive resources, whose access is governed by existing real-time synchronization protocols. The same authors also proposed GPUSync [52], which is a software framework for GPU management in multi-core real-time systems. GPUSync provides support for both fixed- and dynamic-priority scheduling policies. It also provides functionalities such as budget enforcement, multi-GPU support, and clustered scheduling. Based on this synchronization-based approach, [53] extends the analysis proposed in [54] [55] [105] [106], to propose a less pessimistic response-time analysis framework to decide the schedulability of tasksets which may use one or more accelerators. This analysis assumes the use of the Multiprocessor Priority Ceiling Protocol (MPCP) [105], while incorporating the effect of self suspensions [106] [107].

In [51], Kim et al. proposed the server-based approach, where a server task is created to access the GPU on behalf of the client applications, to arbitrate access to a GPU. In this approach, a server task is created which performs access to the GPU on behalf of the client applications. Each task submits its GPU requests to the server and suspends. The server adds each GPU request to a priority queue, and if the GPU is free, it dispatches the highest-priority task in its queue to access the GPU. Once a GPU request completes,

the server wakes up the task which made the request, and copies over the computed result to the task. For this approach, the authors also propose a response-time analysis framework. In this dissertation, we focus on the synchronization-based approach. However, our proposed techniques can be extended to the server approach.

2.3.2 Exploiting Concurrency

All the above-mentioned analyses treat the accelerator as a mutually-exclusive resource and do not consider concurrency. Treating the accelerator as a mutually-exclusive resource leads to increased pessimism in the schedulability analysis. Nevertheless, most modern GPU architectures support the concurrent execution of requests belonging to different tasks.

Consider GPUs from NVIDIA. Each GPU consists of multiple streaming processors (SMs). Typically, embedded GPUs have one or two SMs [22], while high-end GPUs can have 10-20 SMs [108]. In terms of an application, each request is structured as a *kernel*, which consists of a fixed set of parallel instructions. Each kernel consists of multiple threads which combine to form thread blocks, which are assigned to one or more SMs on the GPU. Based on the number of thread blocks in a kernel, it is possible for more than one kernel to be scheduled on a single SM.

A number of techniques have been proposed in the literature to co-schedule task requests and effectively exploit concurrency in GPUs. [109] proposed the Kernelet framework, which dynamically reshapes GPU kernels to improve process throughput. However, Kernelet does not consider task deadlines. In [110], the authors propose S^3DNN which exploits the execution characteristics of DNNs at runtime to optimize their dispatch times, so as to maximize concurrency while meeting task deadlines. In [111], the authors infer the characteristics of the GPU scheduler on the NVIDIA TX1 platform to gain more insight into their use in real-time systems. However, none of the above-mentioned approaches focus on analyzing the schedulability of tasksets which utilize

concurrent accelerators. Specifically, we consider the global-scheduling paradigm, where task requests can be scheduled concurrently on any available fraction of the hardware accelerator.

Recent work has also looked at partitioning the GPU into multiple fractions to which tasks can be assigned. In general, partitioned scheduling can lead to more predictable operation due to less interference from tasks outside the partition. NVIDIA, in their recent GPU architectures, provides the multi-process service (MPS) [35] for partitioning. However, MPS only performs compute partitioning and does not consider memory resources. Therefore, [37] introduces the Fractional-GPU approach which provides both compute and memory partitioning. As a part of this dissertation, we consider partitioning as a tool to make unschedulable tasksets schedulable, by reducing interference between tasks which don't fit well together. However, we consider only computational resources and leave memory for future work.

2.3.3 Energy Savings and Hardware Accelerators

Most accelerators do not provide user-configurable sleep states, and only support voltage and frequency scaling. Thus, sleep-state-based techniques cannot be used in their context. Therefore, the work in [112] proposes a hardware-based approach called MERLOT for GPU energy management in the context of real-time systems. MERLOT exploits the fact that, in the general case, most GPU kernels do not execute up to their worst-case execution time. In such situations, the slack can be dynamically used to reduce the voltage and frequency of the GPU. However, MERLOT considers individual job deadlines, and does not consider taskset schedulability, or the fact that most tasks execute using a combination of CPU and GPU segments. In this dissertation, we instead focus on proposing analytical approaches to compute the best CPU and accelerator frequency pairs to reduce energy consumption, while ensuring taskset schedulability.

Chapter 3

Enabling Time-based Coordination in Cyber-Physical Systems

Reliable *cyber-physical* coordination requires a shared and precise notion of time. As illustrated in Figure 1.1 (in Chapter 1), these time-aware cyber-physical systems have extremely diverse timing requirements. Additionally, even within a cyber-physical system, the timing precision and accuracy requirements of one domain of connected objects may be substantially different from another, and may also change over time. Figure 3.1 illustrates this idea by showing an example system with heterogeneous device types, and communication channels forming different timing subgroups.

While technologies likes GPS and the Network Time Protocol (NTP) [1] have enabled networked devices to share a precise notion of time, trends like networking delays [1], multi-core processors and virtualization [25] introduce greater timing uncertainty. This uncertainty is rarely visible to applications, and most systems rely on best-effort time synchronization. In this chapter, we advocate for timing uncertainty to be visible, controllable and verifiable. To do so, we introduce the concept of *Quality of Time* (QoT), which represents "the end-to-end uncertainty in the notion of time delivered to an application by the system". Building on QoT, we present the QoT Architecture [4], centered around a shared *virtualized* notion of time, which allows applications to specify their tim-

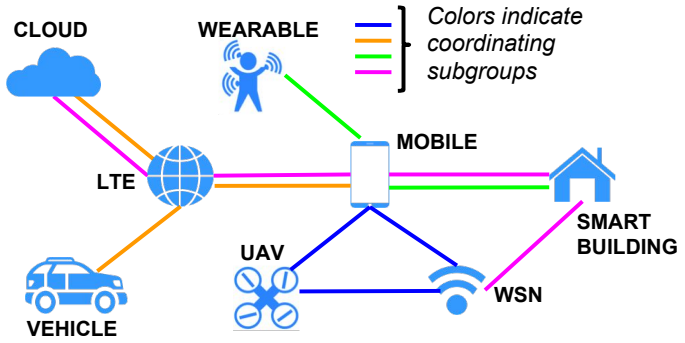


Figure 3.1: Coordinating subgroups in a cyber-physical system require access to a shared sense of time

ing requirements, while delivering the required QoT and exposing timing uncertainty to applications. We argue that the knowledge of QoT enables applications to adapt and be fault-tolerant, while allowing the system to manage resources efficiently.

To achieve these objectives, this chapter also introduces the *timeline* abstraction, which features a *factored-coordination* paradigm for managing time in computing systems. This abstraction enables developers to implement coordinated applications easily. Consider an application that needs to perform coordinated actions by its distributed components. Each of these components bind to a common timeline, each specifying its respective QoT requirements. Application-specified QoT requirements open up the possibility of network and system orchestration to ensure that application requirements are met, while managing resources efficiently. This is fundamentally different from existing best-effort clock-synchronization techniques [1] [2].

The primary contributions described in this chapter are as follows[1] [4]:

- We introduce the notion of Quality of Time (QoT) along with the *timeline* abstraction, which together *re-define* how time-aware applications and the operating system exchange information about time and timing uncertainty.

- We present an application programming interface (API), centered around the *timeline* abstraction, that allows developers to implement time-aware applications.

[1]portions of this work were done jointly with Fatima Anwar, Andrew Symington and Adwait Dongare

- We present the QoT Stack for Linux, a realization of the QoT Architecture illustrated in Figure 3.3. The QoT Stack allows applications to specify their timing requirements, while delivering the required QoT and exposing timing uncertainty to applications.

- We evaluate the capabilities of the QoT Stack on a Linux-based embedded platform, the *Beaglebone Black* [113].

3.1 The Case for Shared Time and QoT

We now argue for designing coordinated CPS using a shared notion of time with the associated knowledge of QoT. In distributed software systems, a shared notion of time enables increased performance and better coordination, along with decreasing the number of messages which need to be exchanged [16]. However, there are inherent uncertainties associated with synchronizing clocks over a network. Hence, in [16], Liskov reasons that systems should rely on clock synchronization for performance but not for correctness. This is true for most software systems. For example, reducing timing uncertainty decreases the transaction commit wait in Spanner, leading to better performance [6]. However, in CPS, the uncertainty tolerances are dictated by the application and the environment. If the required QoT cannot be met, then the application should be aware of it, and gracefully degrade to satisfy safety and reliability requirements.

We highlight the benefits of coordination using a shared notion of time by presenting an emerging CPS application utilizing an *idealized* solution called *TimeNet*. Subsequently, we present the practical challenges in enabling scalable coordination in CPS using shared time and QoT.

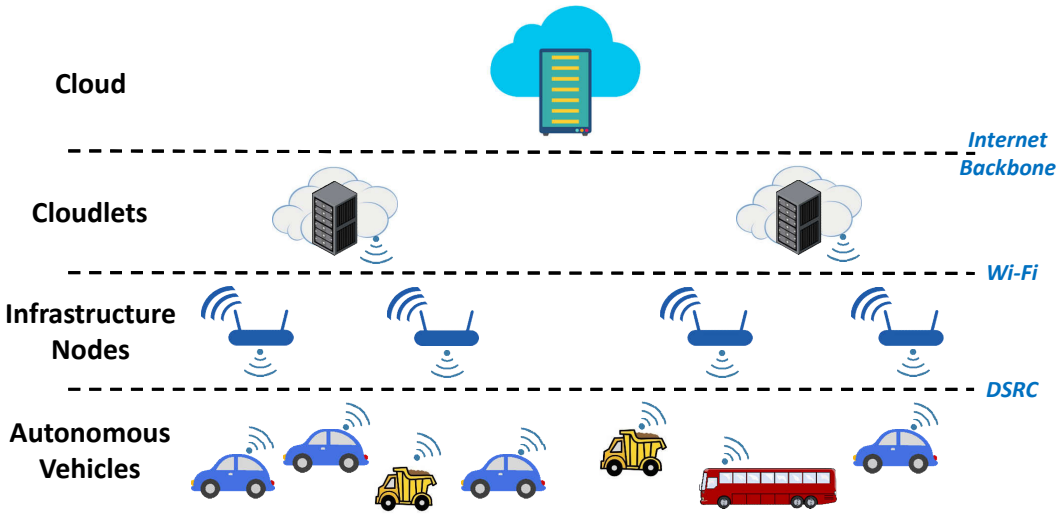


Figure 3.2: Hierarchy of the dynamic traffic-management application. The text on the left highlights the components, while the text on the right indicates the networking technology used to connect adjacent levels of the hierarchy

3.1.1 Connected Vehicles using *TimeNet*

Coordinating fleets of connected autonomous vehicles for city-wide dynamic traffic management is an example of a geo-distributed application which can benefit from using a shared notion of time. The proposed application hierarchy is illustrated in Figure 3.2, and consists of autonomous vehicles, *Vehicle-to-Infrastructure* (V2I) nodes, cloudlets and the cloud.

In an ideal world, we can assume that all components of this application are connected to a network which provides instantaneous access to an *ideal* source of time with no associated uncertainty. For the sake of simplicity, let's call this hypothetical network *TimeNet*. Let's assume that *TimeNet* can be used to perfectly timestamp all events and messages with zero uncertainty. Hence, using *TimeNet*, a unique total ordering on all events can be derived.

In the context of our application, the infrastructure nodes can precisely measure the location of the vehicles, along with the exact timestamp associated with a vehicle's presence at that location. This timestamped information can be then forwarded to a nearby cloudlet, which receives state information from multiple infrastructure nodes in

a small geographical area. Multiple such cloudlets can then forward their respective state information to the cloud, which sits atop the application hierarchy. This provides a snapshot of the traffic conditions in the city to the distributed coordination policies running in the the cloud.

In this hierarchy, the cloud is responsible for shaping traffic flow at a macroscopic level. Based on the macroscopic policy, the cloudlets make local decisions for their respective regions. Lastly, infrastructure nodes decide microscopic traffic policy and convey instructions to the autonomous vehicles, which implement these instructions. At each stage of the hierarchy, important information is distilled and passed on to the next tier, thus reducing the bandwidth demand on the network. Additionally, as microscopic conditions change faster than macroscopic ones, communication frequency reduces and QoT requirements reduces as we go up the hierarchy.

In an ideal world, accurate information can be inferred from these timestamped events, which can be used to formulate plans of action, such that vehicles coordinate their actions using this ideal notion of time. Thus, vehicular traffic is dynamically managed at city scale. In the worst case, if timing constraints are violated or messages delayed, then by using the current time, components can detect failures, and take corrective action [16].

Unfortunately, a perfect source of time does not exist, and practical systems introduce uncertainty in timing measurements. Hence, to determine the validity of timestamps, the knowledge of its associated uncertainty is essential. Based on this uncertainty information, coordination policies can order events with different degrees of confidence. If the uncertainty exceeds tolerable limits, systems can fail-over or gracefully degrade. For example, in the context of the dynamic traffic management application, if the uncertainty exceeds tolerable limits, the coordination policy can instruct all or some vehicles to temporarily change their speeds, or come to a safe halt.

Exposing the notion of QoT to applications also allows timing requirements to be explicitly specified. This enables the system to optimize for application QoT requirements,

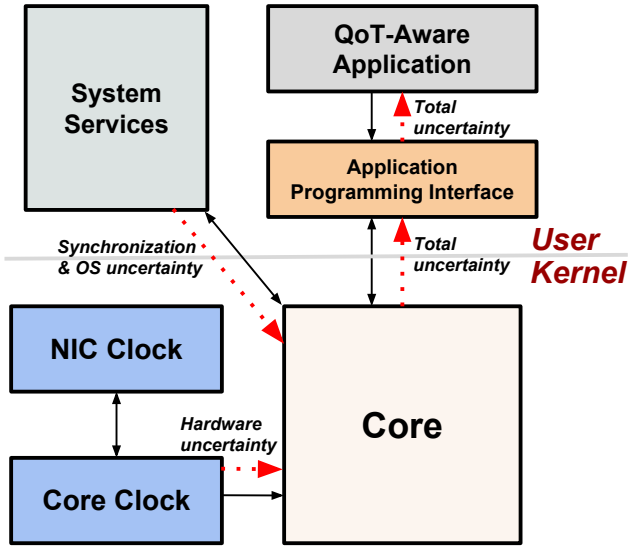


Figure 3.3: Timeline-driven Quality-of-Time Architecture

and manage resources efficiently. Hence, in the context of CPS, synchronized clocks along with QoT can deliver both performance and reliability.

The present-day GPS is a close approximation to *TimeNet*, ideally providing synchronization in the order of tens of nanoseconds. However, GPS is not accesible indoors and inaccurate in urban canyons. This limits its use in many applications. Hence, a practical realization of *TimeNet* may involve multiple outdoor GPS receivers equipped with chip-scale atomic clocks [114]. These receivers can distribute accurate time to subscribers both wirelessly and over the Internet [1]. To support the notion of QoT, it is crucial that each node in *TimeNet* quantify the uncertainty in its notion of time.

Thus, to enable *fault-tolerant* time-based coordination in cyber-physical systems, there is a need for a coordination abstraction which allows a distributed application to specify its QoT requirements to the underlying system.

3.2 Timelines

Modern operating systems keep track of time by means of a single hardware timer driven by an oscillator. Take the case of Linux: multiple virtual clocks, such as

`CLOCK_REALTIME` and `CLOCK_MONOTONIC`, are derived from a single hardware timer. These virtual clocks all share the same accuracy and resolution, and expose themselves to applications in userspace via the POSIX clock [77] interface. This interface allows clocks to be read by an application, and be disciplined using clock-synchronization algorithms such as NTP [1] and PTP [2].

These clock-synchronization algorithms are based on the "trickle-down time" approach, shown in Figure 3.4a, where a static master sits on top of a timing hierarchy. All the other nodes in the hierarchy synchronize their clocks to this *master* and measure time with respect to its reference clock. Furthermore, the master's synchronization rate for the entire network *may be* statically chosen based on the slave with the tightest accuracy requirement. Such approaches are inherently not adaptive and wasteful of resources.

We introduce the *timeline* abstraction, which features an alternative paradigm based on *factored coordination*, where multiple coordinating application components bind to a common timeline to synchronize their clocks to one another, as shown in Figure 3.4b. A timeline provides a shared *virtual* clock reference to all the distributed components of an application. Consider an application that needs to perform coordinated actions at its distributed endpoints. All of these components bind to a common timeline, each specifying its respective QoT requirements. As a result, the timeline abstraction provides the following functionalities:

1. allows an application to specify which components coordinate with each other using *shared time*, and

2. provides visibility into where each application component is deployed, and what its QoT requirements are with respect to the timeline reference.

The above-mentioned functions allow the underlying framework to orchestrate the clock-synchronization protocols and infrastructure to ensure that QoT requirements are met, while making the achieved QoT visible to the application.

(a) Trickle-Down Time Model (b) Timeline Model

Figure 3.4: Traditional v/s Timeline-based synchronization models [4]

Thus, a timeline *abstracts away* clock synchronization from the application. Additionally, a timeline is not necessarily tied to any standard timing reference (such as UTC), and, in the context of distributed coordination, serves as the "narrow waist". This enables developers to easily develop distributed time-based applications on heterogeneous infrastructure, using a timeline-based API. For example, multiple players in the same locality playing a virtual/augmented-reality game need not synchronize to an external time server. Instead, each player can *bind* to a timeline, which synchronizes their clocks to one another, so as to meet their QoT requirements.

The ability of a timeline to expose a virtual clock reference allows different coordinating sub-groups with varying QoT requirements to each have its own time reference and co-exist on the same infrastructure. Note that each node bound to a timeline can have different QoT requirements with respect to the chosen reference. These QoT requirements are generally defined by (i) safety constraints, (ii) performance requirements and/or (iii) the assumptions/tolerances of the controller/decision-making entity. Additionally, multiple *virtual* timelines can coexist on a single node, although they can be tied to a single time source. Thus, timelines overcome the limitations of the static master-slave synchronization paradigm, and support applications that dynamically bind and unbind from timelines in an ad-hoc fashion.

Motivated by the timeline abstraction, the next section introduces the QoT Architecture, along with its corresponding Linux implementation.

3.3 QoT Architecture and Stack

We now present the QoT Architecture, which provides a framework to enable CPS, which rely on a shared notion of time for performing distributed coordination. Using the timeline abstraction, the QoT Architecture takes in application QoT requirements, orchestrates the underlying infrastructure to meet them, and makes the delivered QoT (timing uncertainty) observable to QoT-aware applications.

The QoT Architecture consists of three distinct components:

1) Clocks are used to expose timekeeping hardware, and provide timekeeping and time-stamping capabilities. Clocks also expose their parameters such as accuracy, precision and drift, which enable uncertainty calculations.

2) System Services are responsible for distributing timeline meta-data, message passing, quantifying timing uncertainties, and synchronizing clocks across nodes.

3) The **QoT Core** acts as a bridge between all the system components, applications and the operating system. It is responsible for application scheduling as well as maintaining synchronization and timeline state.

The architectural components present on each node along with their interactions are illustrated in Figure 3.3. Based on this architecture, we developed a prototype QoT Stack for Linux [4], which focuses on implementing necessary functionality over a Local-Area Network (LAN). Along with a wide range of available software, Linux supports platforms ranging from embedded to server-class processors. This makes it an ideal target OS for both prototyping and real-world use-cases. A detailed architectural diagram for the QoT Stack for Linux can be found in Figure 3.5. In subsequent sections, we briefly describe each component of the QoT architecture, and its corresponding implementation in the QoT Stack for Linux. More details can be found in [4].

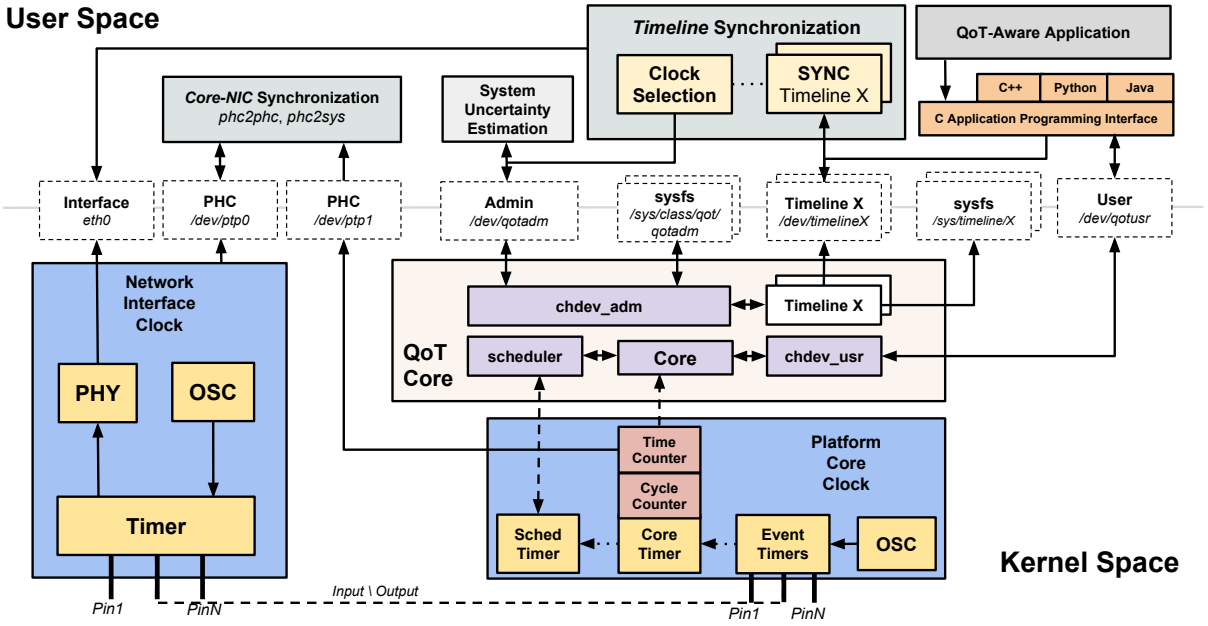


Figure 3.5: The QoT Stack for Linux

3.4 Clocks

The QoT architecture characterizes timekeeping hardware as *clocks*. These clocks play a key role in providing a shared notion of time to userspace applications. Each clock also keeps track of its inherent uncertainty, which is useful for estimating QoT. Based on the functionality provided, the QoT architecture supports two types of clocks:

1) Core Clocks are integral to maintaining a shared notion of time, and all timelines derive their reference time as a projection from the core clock. For a clock to be utilized as a core clock, it must provide (i) the ability to read a *strictly-monotonic* counter, which cannot be modified, (ii) the ability to generate hardware interrupts which can be used to schedule events, and (iii) provide the hardware resolution and uncertainty associated with reading the clock. Optionally, a core clock may also provide the ability to timestamp external events, or precisely trigger hardware events in the future.

2) Network Interface Clocks, also referred to as NICs, assist in disciplining the local timeline clock to some chosen reference time. Modern network interfaces often provide the ability to accurately timestamp network packet transmission and reception at the

physical layer. This can enable more accurate measurements of the propagation delay associated with a medium, which in turn can enable precise calculation of the offset between two clocks. Like a core clock, a NIC also provides the ability to read time, and optionally may provide the ability to precisely timestamp an event, or generate a very deterministic pulse in the future. A NIC however, differs from a core clock in that (i) it is disciplinable and *may* not necessarily be monotonic, and (ii) it does not provide the ability to schedule interrupts. Hence, it cannot be used to schedule user-level application threads. Given these differences in features and functionality, the QoT architecture needs to support both clock types. As described in later sections, this adds an additional requirement for both these clock types to be synchronized with each other.

Linux Implementation: The clocks in our QoT Stack implementation (illustrated as *Network Interface Clock* and *Platform Core Clock* in Figure 3.5) are managed via drivers, and we make use of the Linux `ptp_clock` libraries to abstract away from hardware-platform-specific clock sources. This abstraction provides the ability to (i) enable or disable the clock source, (ii) read the clock source, (iii) configure timer pins (for timestamping inputs or generating pulse-width modulated outputs) and (iv) discipline the external clock – either in hardware or in software. Additional details of the prototype clock drivers implemented for the ARM-based Beaglebone Black [113] can be found in [4].

3.5 System Services

In the QoT architecture, the userspace system services are responsible for distributing timeline metadata, computing QoT, and performing clock synchronization. The primary system services are described as follows:

1) The **Data Distribution Service** is a publish-subscribe standard for real-time systems [115], which we utilize to disseminate timeline metadata across the network. This provides applications visibility into available timelines. Additionally, the system can also use this information to configure and optimize synchronization strategies to meet

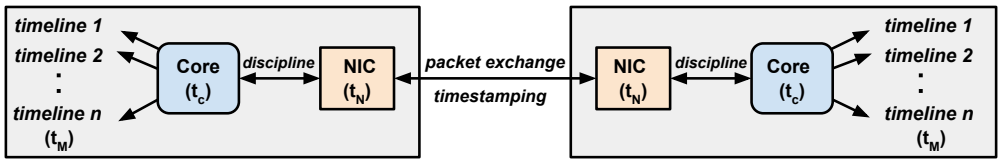


Figure 3.6: Timeline-based end-to-end time synchronization

application QoT requirements. In the QoT Stack, we utilize OpenSplice [115] as the data distribution service.

2) The **Synchronization Service** is responsible for exchanging timestamps to map local time to some reference time. Our timeline-driven architecture supports multiple timelines on a single node, with each timeline maintaining its own notion of time. In the QoT Stack, we maintain each timeline reference as a mapping from a *local* core time to a *global* timeline reference. To generate this mapping, we perform clock-synchronization in two steps as shown in Figure 3.6. The first step performs *intra-node* synchronization by accurately aligning *each* network-interface clock (NIC) to the presiding core clock. By this process, the timestamps provided by a NIC can be considered as equivalent to the core clock. Hence, these NIC timestamps can now be utilized to perform clock synchronization between nodes, using existing clock-synchronization protocols. Additional details about the synchronization-service implementation, and the process by which it estimates QoT can be found in [4].

3) The **System-Uncertainty Estimation Service** estimates the uncertainty introduced by the OS in reading a timestamp. Every timestamp read by a user application contains an uncertainty value introduced by the operating system. This uncertainty varies, and is a function of different factors like the system load and the CPU operating frequency. Therefore, this service continuously updates these uncertainty statistics and passes it to the QoT Stack, which in turn utilizes this value while computing the end-to-end uncertainty bounds.

3.6 QoT Core

The QoT Core (also referred to as the *core*) is central to the QoT architecture and acts as a point of *information exchange* between applications, clocks, system services and the host operating system. The core performs the following key functions:

1) Timeline Management: To satisfy different QoT requirements, the core keeps track of different timelines and the applications bound to these timelines. It also provides an interface for applications to bind/unbind to/from a timeline, as well as specify/update their QoT requirements.

2) Clock Management: The core provides an interface for different hardware clocks to register with it, and exposes an interface for a *privileged* user or service to choose and switch between these different hardware clocks. The core utilizes this chosen clock to maintain a monotonic sense of time, referred to as *core time*. The core also maintains the per-timeline projection parameters from the core clock to each timeline reference, and also provides an interface for the synchronization service to manipulate these per-timeline projection parameters.

3) Event Scheduling: Scheduling an application on a global notion of time is important to execute distributed tasks synchronously. Hence, the core provides applications the ability to *synchronously* schedule events based on a timeline reference. The core provides this functionality in the form of *blocking waits* by interfacing with the operating system scheduler. Note that a *blocking wait* consists of an application suspending, and requesting the OS scheduler to *re-schedule* the application at or after a specified absolute time instant or relative time duration. Our scheduling subsystem is designed to dynamically compensate for any synchronization changes made to a timeline reference. The design of the scheduling subsystem is such that it is agnostic to the scheduling policy followed by the operating system, and only changes the state of an application from *ready* to *waiting* and vice versa. This gives flexibility to operating system designers to optimize the scheduler for different scheduling metrics, based on the target platform.

Table 3.1: Quality of Time APIs

Category	API	Return Type	Functionality
Timeline Association	`timeline_bind` (name, accuracy, resolution)	timeline	Bind to a timeline
	`timeline_unbind` (timeline)	status	Unbind from a timeline
	`timeline_getaccuracy` (timeline)	accuracy	Get binding accuracy
	`timeline_getresolution` (timeline)	resolution	Get Binding resolution
	`timeline_setaccuracy` (timeline, accuracy)	status	Set Binding accuracy
	`timeline_setresolution` (timeline, resolution)	status	Set Binding resolution
Time Management	`timeline_gettime` (timeline)	uncertain_timestamp	Get timeline reference time with uncertainty
	`timeline_getcoretime` ()	uncertain_timestamp	Get core time with uncertainty
	`timeline_core2rem` (timeline, core_time)	uncertain_timestamp	Convert a core timestamp to a timeline reference
	`timeline_rem2core` (timeline, time)	uncertain_timestamp	Convert a timeline reference timestamp to core time
Event Scheduling	`timeline_waituntil` (timeline, absolute_time)	uncertain_timestamp	Absolute blocking wait
	`timeline_sleep` (timeline, interval)	uncertain_timestamp	Relative blocking wait
	`timeline_setschedparams` (timeline, period, start_offset)	status	Set period and start offset
	`timeline_waituntil_nextperiod` (timeline)	uncertain_timestamp	Absolute blocking wait until next period
	`timeline_timer_create` (timeline, period, start_offset, count, callback)	timer	Register a periodic callback
	`timeline_timer_cancel` (timer)	status	Cancel a periodic callback
	`timeline_config_events` (timeline, event_type, event_config, enable, callback)	status	Configure events/external timestamping on a pin

4) QoT Propagation: One of the key functions of the QoT architecture is to expose the end-to-end timing uncertainty to applications. As shown in Figure 3.3, the core helps collect the measured uncertainties from different sources and combines them to compute an end-to-end QoT estimate. These QoT estimates are appended to every timestamp.

In the QoT Stack for Linux, the QoT Core (shown as the central component in Figure 3.5) is implemented as a loadable kernel module. This design choice ensures that no changes are made to the Linux kernel, ensuring portability across different kernel versions. Additional details of the QoT Core kernel module can be found in [4].

3.7 Application Programming Interface

Ease of application development is one of the key objective of the QoT Stack. Therefore, we propose an API that allows application developers to simplify the development of distributed time-aware applications. The key API calls are described in Table 3.1. Based on their functionality, we can categorize the API calls as follows:

1. *Timeline Association* APIs allow applications to bind/unbind to/from a specific timeline, and specify/update their QoT requirements.

2. *Time Management* APIs allow applications to read the timeline notion of time with the uncertainty estimate.

3. *Event Scheduling* APIs allow applications to schedule events using absolute and relative blocking waits on the timeline reference, along with returning the timing uncertainty in when the event was actually scheduled. Additionally, the APIs also provides the ability to trigger events at a deterministic point in the future, as well as accurately timestamp external events, contingent on hardware support from the core clock.

We now present an example *Time Division Multiple Access (TDMA)* application in Listing 3.1, which was written using our C API. To successfully perform TDMA, multiple nodes need to be allocated transmit slots, such that no packet collisions occur. Therefore, it is essential that all nodes participating in the TDMA transmissions have access to a shared notion of time, along with visibility into the associated timing uncertainty. In current implementations of TDMA, the application over-compensates for timing uncertainty by using *guard bands*. However if timing uncertainty increases beyond these guard bands (for example, if synchronization is lost), then packets will collide. Hence, providing the application with uncertainty measurements by means of QoT bounds, enable the application to *adapt* when the required QoT cannot be delivered.

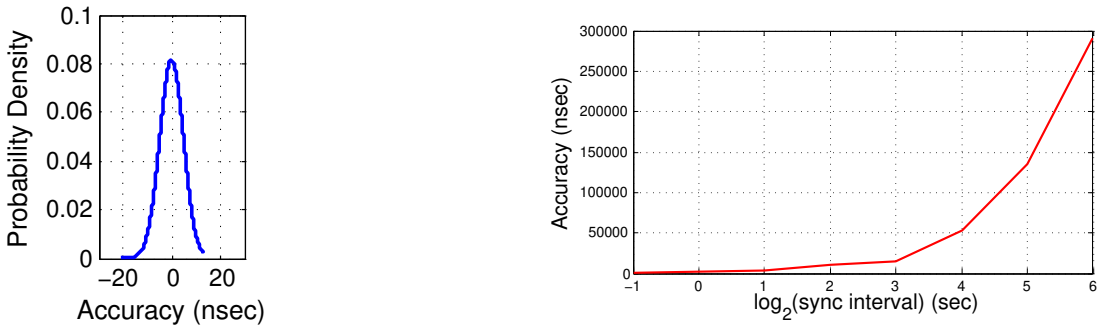
The TDMA application described in Listing 3.1, starts by creating a binding to a timeline, with desired accuracy and resolution using `timeline_bind`. Given that transmitting in a TDMA slot is inherently periodic, the application sets its period and start offset using `timeline_setschedparams`. Subsequently, the application executes a loop, where it calls `timeline_waituntil_nextperiod`, which wakes the task up every period, using the programmed period and start offset. This call also returns an uncertainty in the time when the scheduler returned control to the application. The application can make use of this information to take a decision on transmitting a packet. Finally, before the application terminates, it unbinds from the timeline using `timeline_unbind`.

Listing 3.1: QoT-aware TDMA Application

```
/* Binding Parameters */
timeinterval_t accuracy;
accuracy.below = {0, 1e12};          /* 1 microsecond */
accuracy.above = {0, 1e12};          /* 1 microsecond */
timelength_t resolution = {0, 1e9}; /* 1 nanosecond  */
period = TDMA_CYCLE;
start_offset = get_my_slot ();
name = "tdma-timeline"
/* Bind to a timeline with requested UUID */
timeline_t timeline;
timeline = timeline_bind (name, accuracy, resolution);
/* Set period and start offset */
timeline_setschedparams (timeline, period, start_offset);
/* Transmit Packets using TDMA slot */
while (tdma_running) {
  timestamp = timeline_waituntil_nextperiod (timeline);
  if (timestamp.uncertainty < accuracy ) {
    transmit_packet ();
  }
  else {
    hold_off ();
  }
}
/* Unbind from a timeline */
timeline_unbind (timeline);
```

3.8 Experimental Evaluation

We now evaluate the performance of the QoT Stack for Linux. Our prototype implementation provides hardware support for the ARM-based Beaglebone Black (BBB) [113] embedded platform. Therefore, our testbed comprises of multiple BBB nodes connected by means of an IEEE-1588-compliant switch [116].



(a) Core-NIC

(b) Accuracy v/s Synchronization Interval

Figure 3.7: (a) Core-NIC synchronization accuracy (b) Illustrating the adjustable synchronization parameter

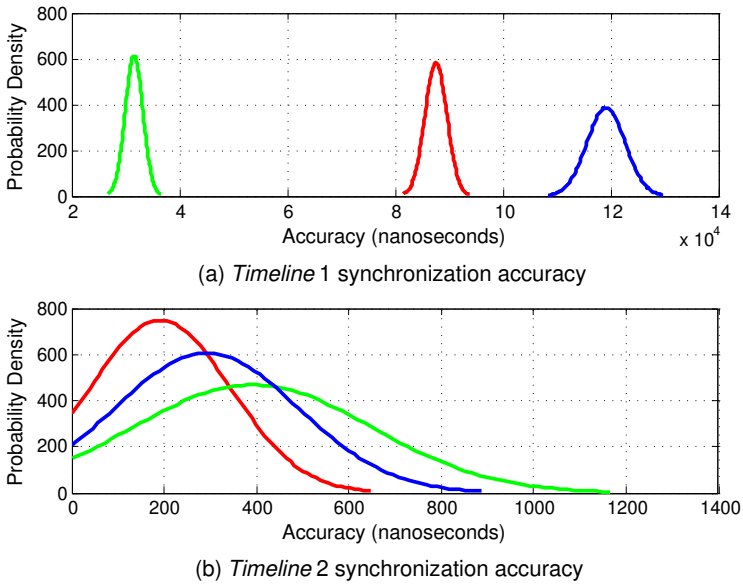
3.8.1 Clock-Synchronization Measurements

We now describe the evaluations performed to benchmark the clock-synchronization capabilities of our QoT Stack.

Core-NIC Clock Synchronization: As described in Section 3.5, to achieve end-to-end synchronization, i.e., mapping *local* core time to a *global* timeline reference, the first step involves *accurately* synchronizing the on-board network-interface clock (NIC) with the local core clock. Figure 3.7a plots the probability distribution of Core-NIC synchronization accuracy. Observe that we achieve an accuracy in the order of tens of nanoseconds by utilizing a hardware-programmable timer on the BBB AM335x. This timer deterministically triggers periodic voltage-change outputs on a pin, which is then timestamped by the NIC, to work out the clock-disciplining parameters.

Tunable Clock Synchronization: The ability to adapt to application QoT requirements is also a key proposition of the QoT architecture. Therefore, we investigate the use of modifying the transmission rate of synchronization packets to tune the clock-synchronization accuracy. Figure 3.7b plots the measured clock-synchronization accuracy as a function of the synchronization interval (inversely proportional to synchronization rate). Observe that, as we increase the synchronization-packet transmission rate, the synchronization error reduces.

End-to-End Clock Synchronization: We consider a topology similar to the one il-

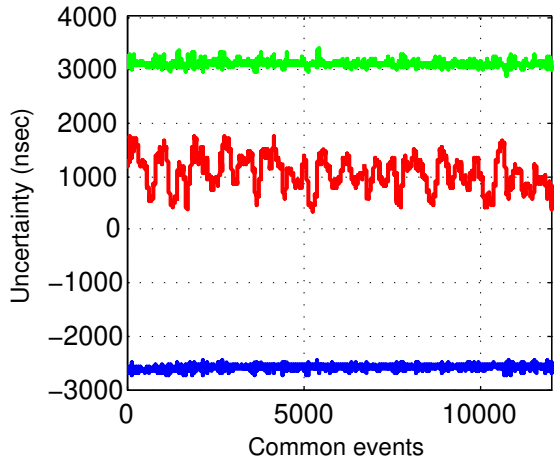


(a) *Timeline* 1 synchronization accuracy

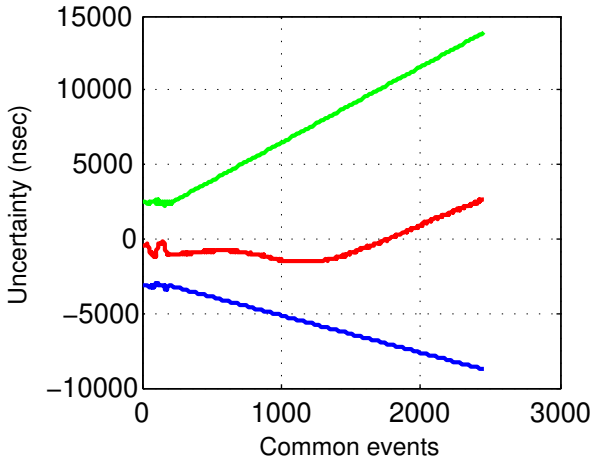
(b) *Timeline* 2 synchronization accuracy

Figure 3.8: (a) shows pair-wise error probability density of three nodes a, b, c bound to Timeline 1 in Figure 3.4b with 100 μsec accuracy requirement, while (b) shows pair-wise error probability density of three nodes c, d, e bound to Timeline 2 with 1 μsec accuracy (Note that x-axis units are in nanoseconds, and x-axis scale changes in (a) and (b)). Note that c maintains mappings of both timelines, and the achieved accuracy for all the nodes is almost equal to their desired accuracy

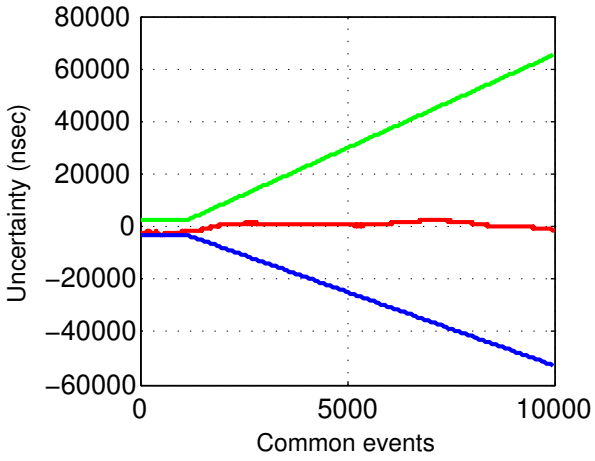
lustrated in Figure 3.4b (Section 3.2) to measure the end-to-end clock-synchronization accuracy. In particular, we consider two timing subgroups: a test application A deployed on nodes a, b and c bound to Timeline 1 with an accuracy requirement of 100 μsec; and a test application B deployed on nodes c, d and e bound to Timeline 2 with an accuracy requirement of 1 μsec. The system sets a synchronization rate of 0.05 Hz for Timeline 1, and 2 Hz for Timeline 2 based on their respective application-specified accuracy requirements. We utilize this topology to demonstrate that the QoT Stack can run multiple parallel synchronization sessions on a single node, which simultaneously disciplines multiple timelines. The results are illustrated in Figure 3.8, where node c maintains two timelines with different accuracy requirements of 100 μsec and 1 μsec, with respect to Timelines 1 and 2 respectively. This experiment validates our claim that the timeline-driven architecture not only supports multiple virtual time references on a single node, but is also able to adapt to meet application QoT requirements.



(a) Synchronization on for 1 hour



(b) Synchronization on, then off for 5 mins



(c) Synchronization on, then off for 1 hour

Figure 3.9: Upper bound (upper green plot) and lower bound (lower blue plot) around the actual uncertainty (middle red plot) with and without synchronization. Note the change in y-axis scale which is increasing from (a) to (c)

QoT Estimation: Figure 3.9 showcases the QoT Stack's ability to estimate valid QoT estimates. These QoT estimates capture the timing uncertainty introduced by different sources of errors, which cause a node's time estimate to diverge from its true value. In Figure 3.9, the red plot provides the ground truth i.e, the actual measured offset between the *local* timeline reference and the *global* timeline reference. On the other hand, the green and blue plots describe the upper and lower QoT bounds respectively, as estimated by the QoT Stack. Note that these provided bounds are valid as they always

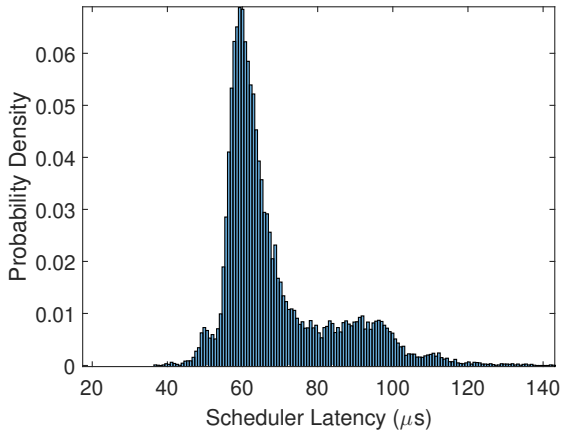
bound the measured error. Section 3.1 highlighted the benefits of QoT for applications to detect clock-synchronization failure. Therefore, Figure 3.9b and 3.9c first synchronize the clock, and then simulate clock-synchronization failure by disconnecting the network. Observe that, after the network disconnects, the provided QoT bounds extend in both directions as a function of variance in frequency bias, and they always bound the actual measured offset between the two nodes. Thus, we can conclude that the QoT Stack provides accurate QoT estimates.

3.8.2 Scheduler Measurements

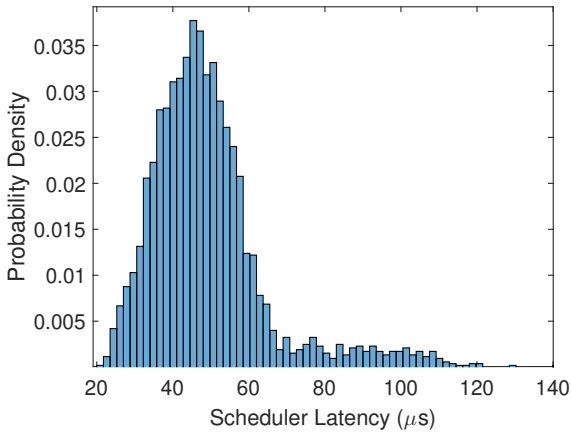
We benchmark the QoT Stack's scheduling interface against the Linux *Real-Time* (RT) scheduler by using periodic pin-toggling applications. All the following experiments were conducted under identical load conditions, for a duration of 3000 seconds, with the pin-toggling application being the highest real-time priority user application in the system. Multiple sporadic tasks with lower real-time priorities, which used the QoT Stack, were also running on the same system.

Scheduler Uncertainty: To measure scheduler uncertainty, we devise the following experiment. On a single node, an application periodically calls the `timeline_waituntil_nextperiod` API call, such that the task is scheduled to toggle a memory-mapped GPIO pin at every second boundary on a timeline reference. When the task wakes up, the QoT Stack provides a timestamp (with uncertainity) for when the event was actually scheduled. The scheduler latency can be estimated by taking the difference of the timestamps: when the task was supposed to wake up, and when it was actually scheduled. We also empirically measure the scheduler latency by using a Salae Logic Pro 16 logic analyzer [117]. The logic analyzer measures the latency for each pin toggle event by comparing against a deterministic PWM with edges at every second boundary on a timeline reference.

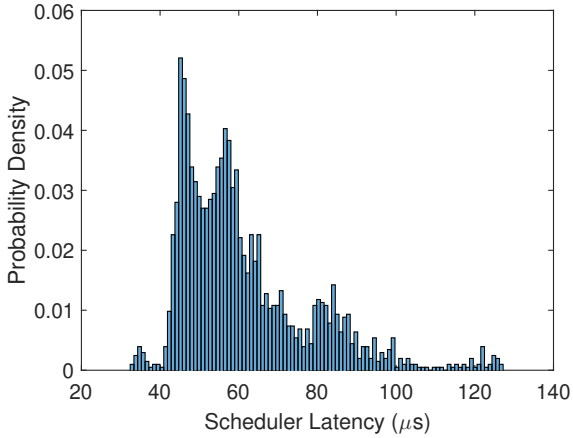
Figure 3.10a plots the distribution of the scheduler latency as estimated by the QoT



(a) Estimated QoT Scheduler Latency



(b) Measured QoT Scheduler Latency

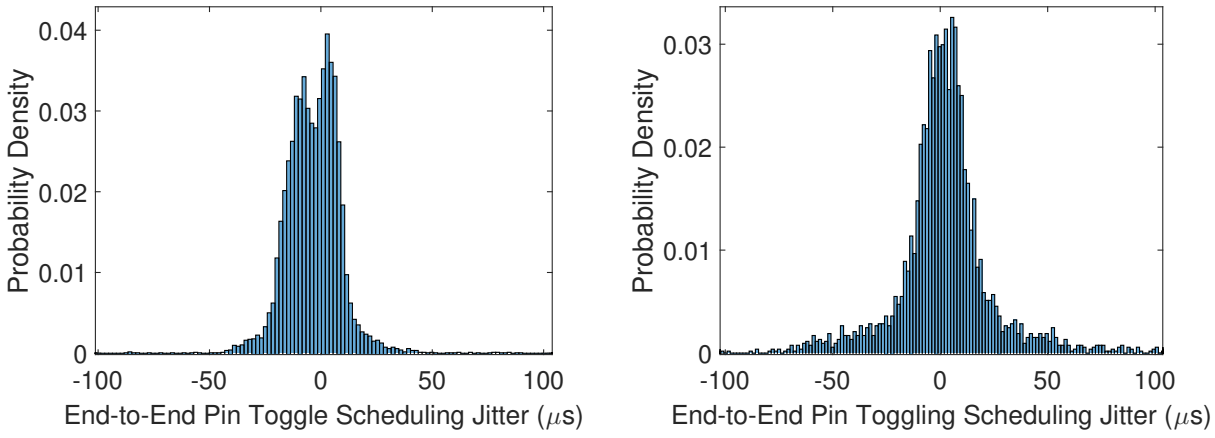


(c) Measured Linux RT Scheduler Latency

Figure 3.10: Scheduler Latency Distributions, for a periodic pin-toggling application on a single node

Stack, while Figure 3.10b shows the empirically-measured distribution. Observe that the empirically-measured distribution and the distribution provided by our stack share similar characteristics. This demonstrates that the uncertainty estimate provided by the QoT Stack holds up to empirical measurement.

For the Linux RT scheduler, using the `SCHED_FIFO` real-time priority scheduling policy, Figure 3.10c shows the measured latency distribution, where the `clock_nanosleep` system call was used to schedule a periodic pin toggle. Note that the QoT-aware Linux scheduler and the Linux RT scheduler share similar statistical properties. The QoT-aware scheduler provides adherence to our *timeline*-driven architecture, with no loss in



(a) QoT Stack - 1 μs sync accuracy (b) Linux RT Scheduler and PTP

Figure 3.11: End-to-End Scheduling Jitter Distributions, for a distributed synchronous pin-toggling application deployed on two nodes

performance.

Coordinated Scheduling: The ability to perform choreographed scheduling is key to our stack, and hence we characterize the end-to-end synchronous scheduling jitter. In our setup, we have two identical applications running on separate nodes. Both applications bind to the same timeline, specifying a synchronization accuracy requirement of 1 μs. The applications synchronously toggle a GPIO pin, using the `timeline_waituntil_nextperiod` API call, at every second boundary on the timeline reference. The synchronization service is also running on both nodes. In Figure 3.11a, we plot a distribution of the end-to-end jitter between the pin toggles of the distributed application. The instants at which the pins were toggled was captured by a logic analyzer, and the difference in timestamps, were used to compute the obtained distribution.

We conduct a similar experiment using the Linux `clock_nanosleep` system call on two distributed nodes synchronized by PTP. Figure 3.11b plots the distribution of the end-to-end scheduling jitter for Linux and PTP. Our stack runs a patched PTP synchronization service. Hence, the distribution obtained has a similar jitter profile to that obtained using PTP. Note that our stack does not suffer any performance loss, while at the same time providing a range of QoT-based functionality.

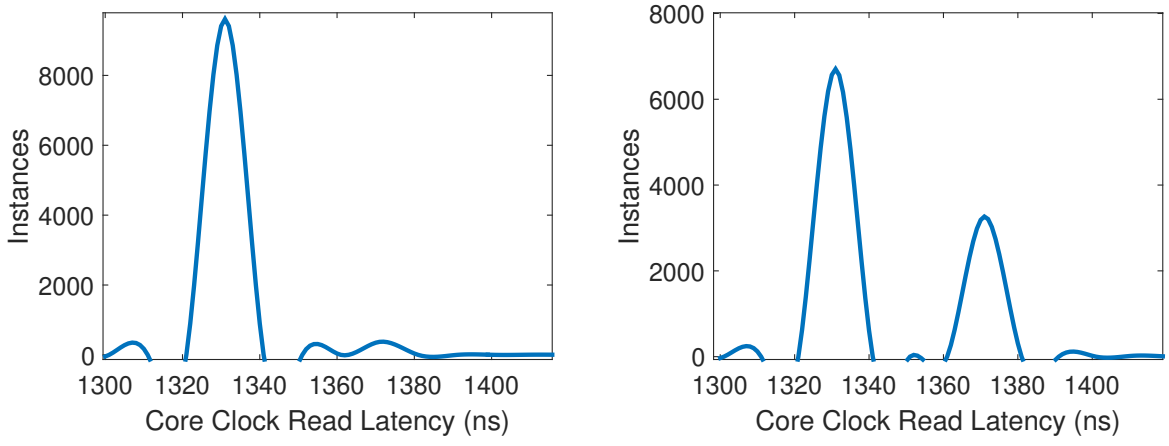


Figure 3.12: Clock read latency histograms in different time intervals, estimated by the system uncertainty estimation service

Clock-Read Latency: Figure 3.12 shows two histograms for the estimated latency in reading the core clock from userspace, over different one-second durations, as estimated by the system uncertainty estimation service. Observe that the distributions change over time and is a function of system load. Each peak in the distribution corresponds to different *locks* which cause contention in reading the core clock. This measured distribution plays a key role in continuously keeping track of the uncertainty introduced by the OS in reading the clock.

3.9 Summary

In this chapter, we introduced the notion of *Quality of Time* (QoT), which represents "the *end-to-end* uncertainty bounds corresponding to a timestamp, with respect to a clock reference." Adopting this *holistic* notion of Quality of Time (QoT), which captures clock metrics such as resolution, accuracy, and stability, we propose an architecture in which the local perception of time is a controllable operating system primitive with observable uncertainty, and where an adaptive clock-synchronization service balances applications' timing demands with system resources such as energy and bandwidth. Our architecture

features an expressive application programming interface that is centered around the notion of a *timeline* – a virtual temporal coordinate frame that is defined by an application to provide its distributed components with a shared sense of time, with a desired *accuracy* and *resolution* – that enables developers to easily write applications whose activities are choreographed across time and space.

Leveraging open-source hardware and software components, we built an implementation of our proposed QoT Architecture called the *QoT Stack for Linux*, and present results from its evaluation. The QoT Stack manages clocks and synchronization protocols to deliver application-specified levels of QoT. Additionally, it also makes the delivered QoT visible so that QoT-aware applications can adapt if the delivered QoT exceeds application requirements. Our QoT Stack for Linux is open-source, and the code can be found at https://bitbucket.org/rose-line/qot-stack/src.

Chapter 4

Bringing QoT to Virtual Machines

To enable scalable time-based cyber-physical coordination, it is essential that we engineer a QoT-aware cloud/edge-cloudlet infrastructure [17]. However, to maintain application isolation, most public clouds and cloudlets provide multi-tenancy using *virtualized* units of computing. These maybe Virtual Machines (VMs) [26] or application containers [27]. Additionally, the use of virtualization for consolidation of multiple real-time systems on a single platform is also of increasing interest [118]. Motivated by these needs, this chapter focuses on bringing the notion of QoT to the dominant virtualization technology, namely virtual machines. We design and implement the QuartzV extension to the QoT Stack for Linux for introducing the notion of QoT to Linux VMs running atop the open-source QEMU-KVM [57] hypervisor.

The contributions described in this chapter are as follows:

- Elucidating the challenges and subsequent architectural choices in bringing QoT to Virtual Machines,
- Introducing the QuartzV extensions for Linux VMs which support para-virtual clocks,
- Porting the QoT Stack for Linux to VMs and hypervisors which do not support para-virtual clocks, and

- Evaluating and comparing the performance and scalability of the para-virtual QuartzV approach against the native and fully-virtualized scenarios.

4.1 Background

We now introduce the background relevant to time and virtualization.

4.1.1 Virtualization

Virtualization is often used to share physical hardware resources among multiple users, while providing the illusion that every user has access to his/her own machine [119]. To support this illusion, it is important that (i) virtualized units are well-isolated from other users [119], and (ii) the overhead of virtualization is low [119]. These objectives are often conflicting, and virtualization technologies generally trade off one of the objectives in favor of the other. For example, hypervisor-based virtual machines [57] [120] offer strong isolation by trading off some performance due to the overhead of the hypervisor. On the other hand, operating-system level virtualization [27] (also known as *containerization*) trades off some level of isolation for performance by eliminating the hypervisor.

In this chapter, we focus on hypervisor-based virtual machines (VMs). Modern hypervisors generally take advantage of *hardware-accelerated* virtualization, based on hardware extensions like Intel VT-x [121] and AMD-V [122]. These technologies enable VMs to execute unprivileged CPU instructions natively, while privileged instructions are serviced using the trap and emulate mechanism [121]. On the other hand, *para-virtualization* [119] enables low-latency access to peripherals and I/O devices, such as network interfaces, disks and clocks, also delivering near-native performance. This access is made possible by para-virtual drivers [119], which can directly perform protected access to the hardware through the hypervisor. For systems which do not support hardware acceleration or for VMs which lack para-virtual drivers, all CPU instructions or peripheral-device access must be emulated in the hypervisor. This is also referred to as

full virtualization [119]. In practice, modern hypervisors generally utilize a mixture of para-virtualization and hardware-accelerated virtualization [57] [120] to provide near-native levels of performance.

4.1.2 Time and Virtualization

The use of hypervisor-based VMs introduces an additional abstraction layer between applications and the hardware. This translates to additional timing uncertainty, due to higher jitter in clock-read and interrupt-servicing latencies [25]. Therefore, in [25], the authors experimentally characterize the timekeeping properties of the Xen hypervisor [119]. Their work highlights the weaknesses of the existing timing solution in Xen, which uses independent NTP [1] synchronization sessions for each guest VM. They refer to this as the *independent clock* paradigm, where each VM independently performs clock synchronization. The authors note that this practice is wasteful of system resources, and degrades synchronization accuracy. Additionally, the authors also find the practice of keeping clock-synchronization state in the VM detrimental for live migration. Hence, the authors propose a *dependent-clock* solution based on the *RADclock* [123] feed-forward synchronization algorithm. Each VM has a dependent clock, which is sourced from the hardware clock on the host machine. Hence, each VM has access to the para-virtualized hardware clock exposed in the host OS. This clock is disciplined in the host OS, and thus only one synchronization service is required per host machine. Apart from being resource-efficient, as VMs now do not contain synchronization state, the dependent-clock paradigm also aids VM live migration. Hence, if a VM is migrated, it need not re-synchronize its clock, and can derive the time from the hypervisor at the new host.

The authors in [25] conclude that the para-virtualized dependent clock is useful for VMs. However, the authors do not consider the utility of exposing timing uncertainty information. Additionally, the recent evolution of hypervisors and the advent of hardware-accelerated virtualization offers a fresh opportunity to re-visit the problem of time and

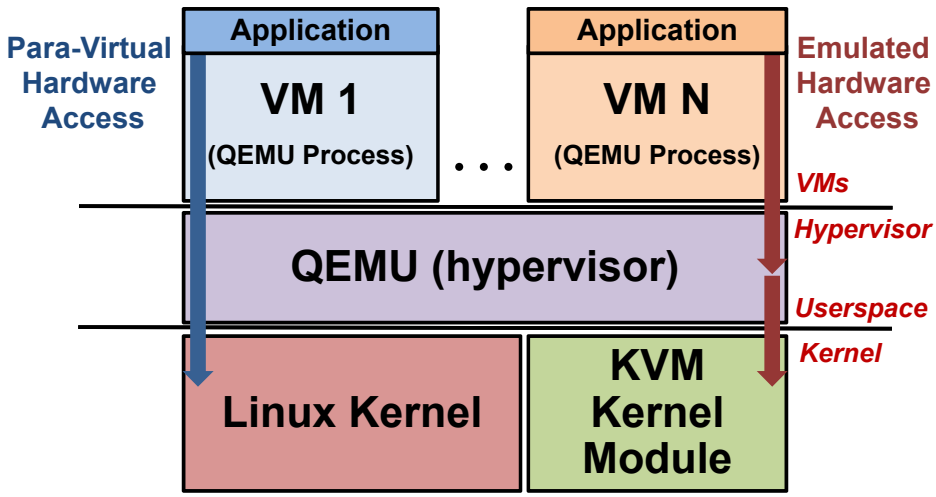


Figure 4.1: The QEMU-KVM Hypervisor. VM 1 supports para-virtual peripheral access, while VM N utilizes peripheral emulation (full virtualization)

virtualization.

4.1.3 Kernel-Based Virtual Machine (KVM)

We focus on one commodity open-source hypervisor, namely Kernel-based Virtual Machine [57], also referred to as QEMU-KVM. However, the core concepts of this work are applicable to other commercial and open-source hypervisors. Figure 4.1 shows the organization of the QEMU-KVM hypervisor, and illustrates how virtual machines interact with its components. QEMU-KVM consists of two core components:

1) The **QEMU Emulator** functions as a hypervisor, and each VM runs as a QEMU process. QEMU can be used for full virtualization (all instructions emulated), or hardware-accelerated virtualization (only privileged instructions emulated). In addition, QEMU also provides VMs with para-virtualized access to host peripherals (such as disks, I/O devices, network interfaces and clocks). For VMs which do not support para-virtualization, QEMU also provides peripheral-device emulation.

2) The **KVM Loadable Kernel Module** enables QEMU to interface with the Linux kernel. This allows QEMU-KVM to use existing kernel functionality for resource management (such as scheduling and isolation). Additionally, the KVM kernel module also

enables the hypervisor to take advantage of hardware extensions like Intel VT-x and AMD-V.

In terms of clock support, QEMU-KVM provides the para-virtual `KVM-clock` [124] to Linux VMs. This allows a para-virtual guest VM to access the host's monotonic system clock (`CLOCK_MONOTONIC`) and real-time clock (`CLOCK_REALTIME`). On the x86 architecture, `KVM-clock` uses the `Time-Stamp Counter` (TSC) [125], and a memory page mapped into the VM's virtual-memory space to provide low-latency clock reads. Whenever the VM is scheduled, the hypervisor writes the current time (monotonic and real-time), and corresponding TSC value into this page. The VM can then use this timestamp along with reading the current TSC value to calculate the current time. Given that both reading the TSC (`rdtsc`) and accessing a memory address are non-privileged operations, a para-virtual guest VM can perform low-latency clock reads. For VMs which do not support para-virtualization, QEMU-KVM provides access to emulated timers [57].

4.2 Time-Based Applications using QoT

Before describing QuartzV, we motivate its utility by describing an application which can be enabled by using virtualization and QoT. Although the application described is from the industrial-automation domain, the core concepts can be adapted for other coordinated distributed application domains.

Consider an industrial-automation application, where multiple robotic arms are used to collaboratively assemble a mechanical assembly. Collaborative manufacturing is often required to: (i) speedup assembly (e.g. performing parallel assembly), and (ii) perform joint tasks which may be too large for a single robot to operate on (e.g. cooperatively picking and placing a large part onto the main assembly). To successfully perform collaborative manufacturing, we need to ensure that the robotic arms are coordinated such that, (i) when performing parallel tasks, they do not interfere with the proper functioning of each other, or operate in/on the same physical space, and (ii) when performing

joint tasks, they coordinate their actuations (actions) to successfully complete the task. While these coordination scenarios can be carried out using extensive message passing, the overhead involved is high. This message-passing overhead prevents a system from scaling to multiple endpoints. Often, industrial systems are over-engineered or hard-coded to achieve such tasks, which limits the capabilities and flexibility of the system.

An alternative approach is to use a shared notion of time as a primitive for coordination [16] [4]. In this case, an intelligent centralized/distributed task planner with a view of the entire system can dynamically generate action commands with a corresponding action timestamp, based on shared time. The endpoints of the system can then execute these actions at the planned time points. However, given that industrial systems are often safety-critical, a fault-detection primitive such as QoT is needed to handle the case of clock-synchronization failure [17].

By specifying the required QoT, each component in the system knows the maximum level of uncertainty tolerable to perform successful coordination. Since each node independently maintains its own notion of QoT with respect to the reference, a node can enter a graceful-degradation [126] mode when the level of uncertainty exceeds the tolerable limit. Additionally, if a coordination message is delayed or arrives too late, all a node needs to do is compare the action timestamp against the current time on its local clock [16]. Based on this timestamp, the endpoint can adapt or enter a graceful degradation mode. Given that modern oscillators drift slowly, the probability of clock-synchronization failure is much lower than the probability of CPUs, networks or disks failing [6]. Therefore, using a shared notion of time with QoT can enable scalable and fault-tolerant coordination [17].

Based on the philosophy of QoT, Figure 4.2 illustrates a collaborative-assembly scenario using two robot arms. The system consists of (i) a centralized task planner running in a VM (using QuartzV) hosted on a server machine, (ii) two embedded-grade arm controller nodes (using the QoT Stack for Linux), and (iii) two robot arms with sensors (to determine state), an end effector, and real-time motor controllers.

We now describe this system in a top-down fashion:

1) The **Task Planner in a VM** is able to receive timestamped sensory input from the sensors on the robotic arms, and can be based on techniques including signal-processing, machine learning [127], or model-based artificial intelligence [128]. Based on the system objective, sensory inputs, and the state of the system, the task-planner can generate receding-horizon-based [129] timestamped action commands for the robotic arms to perform the collaborative assembly. In the context of the example application, the action commands can be in the form of (i) the position of the end-effector, and (ii) "pick" or "place" actions of the end-effector. These action commands are received by the "controller" nodes.

2) The **Controller Nodes** are responsible for generating a time-parametrized feasible motion plan for their respective robotic arms based on the action commands, and the sensory inputs from the arm. This requires converting the coarse-grained end-effector trajectory into feasible fine-grained joint-motion trajectories or end-effector actions, such that collisions are avoided. These embedded controller nodes also contain I/O ports which enable them to directly interface with their respective robotic arms with low latency.

3) The **Robot Arms** contain on-board low-level real-time motor controllers which are responsible for carrying out the motion plan received from their respective controller nodes.

In the described system, the task-planner node (VM) and the controller nodes are inter-connected using a switched Ethernet network. All these three nodes use the QoT Stack functionality to bind to a common timeline with their specified QoT requirements (+/-1 ms for the task-planner node, and +/-100 μs for the controller nodes). The synchronization service (based on PTP [2]) can then discipline the clocks to meet the specified QoT requirements. Notably, there is no need for the robot arms to directly join the timeline. This is because the on-board motor controllers of the robot arm can perform real-time control with deterministic latency [130] [131]. Additionally, sensor values can

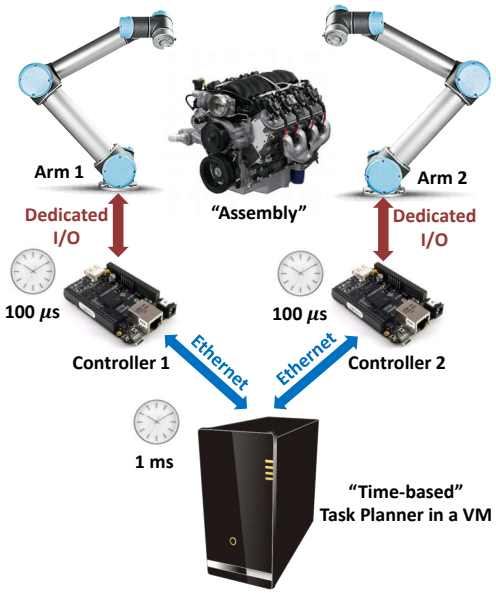


Figure 4.2: Time-based Coordinated Industrial Automation

also be accessed with deterministic latency. This assumption holds true for most industrial robots. Thus, due to the presence of dedicated controllers and interfaces, the robots can carry out the motion plan specified by the embedded controller in deterministic fashion. Additionally, using the dedicated I/O interface, the embedded-controller node can read the robot's sensors with deterministic latency, and hence assign timestamps using its own local timeline reference.

Our objective is to use QuartzV in the design of scalable and fault-tolerant coordinated applications, like the above, using a shared notion of time and QoT.

4.3 QuartzV Extension to the QoT Stack

In this section, we describe the design choices involved in bringing QoT to hypervisor-based virtualization. Subsequently, we present the QuartzV extension to the QoT Stack for Linux to provide QoT awareness for para-virtual guest VMs running atop the QEMU-KVM hypervisor. QuartzV adds extensions to the QEMU-KVM hypervisor, in order to provide *clock-synchronization-as-a-service* to para-virtual guests. This enables applications running in a guest VM to specify their QoT requirements, while a host service tries to

meet the specified requirements, and feeds back the achieved QoT to the application running in the guest VM.

We first discuss the applicability of QuartzV in a para-virtualized setting, and later discuss how our implementation works in an environment where clocks are fully virtualized (emulated), or the hypervisor cannot provide clock-synchronization extensions. While QuartzV has been implemented for QEMU-KVM, the concepts are readily applicable to other commodity open-source hypervisors like Xen [120].

4.3.1 QoT and Virtualization

While designing QuartzV for a para-virtual setting, the following design considerations need to be taken into account:

1) Specifying QoT Requirements: To provide QoT awareness in the virtualization context, applications need to be able to specify their QoT requirements. Hence, we need to develop a mechanism to allow applications running in a VM to convey their QoT requirements to a service running on the host OS. Since specifying QoT requirements is not on the critical path of most applications, we can afford a somewhat higher-latency communication mechanism for this purpose.

2) Exposing QoT to Applications: To expose the notion of QoT to applications, every timestamp read should contain its associated uncertainty. Reading timestamps from a clock is often on the critical path of most applications. Hence, we must provide a timestamp along with its associated uncertainty, with low latency. For this purpose, we require an efficient low-latency mechanism which can transfer a timestamp, along with the achieved QoT from the host to the guest VM.

3) Supporting Multiple VMs: The key idea of virtualization is to consolidate multiple VMs on a single physical machine. Hence, it is imperative that our implementation scale to support multiple VMs without any impact on performance.

4) Maintaining VM Isolation: While workload consolidation is key, isolation be-

tween different VMs is essential. Hence, our implementation should prevent malicious VMs from affecting the correct operation of other VMs.

5) **Portability:** We aim to implement our system such that no modification is made to the hypervisor source code. Instead, we use existing hypervisor functionality to implement our extensions. This ensures that our implementation is portable across different versions of the QEMU-KVM hypervisor.

4.3.2 QuartzV: Design and Implementation

Based on these considerations, we now present the design of QuartzV, which builds upon the previously described QoT Stack for Linux [4] (Chapter 3), to bring the notion of QoT to VMs. The key components of QuartzV are as follows:

1) **QoT Application Library:** Also known as `qotlib`, it provides QoT-specific functionality to user-space applications. This library exposes timeline-based distributed coordination APIs, that are independent of the platform and OS. The APIs enable applications to (i) bind/unbind from a timeline, (ii) specify/update their QoT requirements, (iii) schedule events based on shared time, (iv) timestamp events, and (v) support publish/subscribe messaging for coordination [17]. All API calls return the QoT actually delivered to the application, providing the ability to adapt to changes in QoT [4]. This library can be configured with a compilation flag to enable para-virtual guest-related functionality. This allows native applications to be ported to a VM without any changes to the source code.

2) **QoT Core Kernel Module:** It acts as a bridge between the components of the QoT Stack for Linux [4], and is responsible for timeline management, clock management and time-based event scheduling. Applications and system services interact with the QoT Core using an `ioctl` interface exposed over the `/dev/qotusr` character device. Both the host and guest VMs contain their own QoT Core module. The QoT core's scheduling interface is policy-agnostic [4], and is responsible for moving tasks from the scheduler

wait-queue to the run-queue at the specified time. This provides developers the flexibility to choose the appropriate real-time scheduling policy based on application priorities and requirements.

3) **QoT Clocks:** These are useful for maintaining a shared notion of time, and also aid in performing clock synchronization over a network [4]. The core clock [4] is used to maintain a monotonic free-running (drift not disciplined) notion of time. Each timeline-reference clock `/dev/timelineX` (where X is the timeline id) is mapped from the core clock (on the host) using the parameters tl_{skew} (drift correction), $core_{last}$ (the core-clock timestamp at the last synchronization event) and tl_{last} (timeline-reference timestamp at the last synchronization event). Using the current core timestamp, $core_{now}$, the timeline-reference time, tl_{now}, can be projected as follows:

$$tl_{now} = tl_{last} + tl_{skew} * (core_{now} - core_{last}) \tag{4.1}$$

4) **Synchronization Service:** This is deployed on the host, and synchronizes the local timeline clock, derived from the local monotonic clock source, with the global timeline reference. We use feed-back synchronization to discipline the clock on a per-timeline basis. This service polls the timeline clock over `/dev/timelineX` (where X is the timeline id) to detect any updates to application QoT requirements. Based on these application requirements, the service disciplines the timeline-reference clock by modifying its parameters (drift and offset) to achieve the desired levels of QoT. In doing so, the synchronization service periodically updates the clock mapping parameters and associated timestamp uncertainty lower and upper bounds, ϵ_l and ϵ_h, on a per-timeline basis.

5) **Inter-VM Shared-Memory Server:** Also known as `ivshmem_server`, this is deployed on the host, and creates a POSIX shared-memory region which is used to distribute clock parameters and timestamp-uncertainty information to applications running in guest VMs.

6) **QoT Virtualization Service:** Also referred to as `qot_virtd`, this service is deployed on the host, and aggregates application-specific QoT requirements from different

guest VMs hosted on the host machine. It creates a Unix socket, and acts as a server, while the guest VMs are its clients. Applications running in a VM can send their timeline information and QoT requirements to `qot_virtd` using the created socket. Additionally, whenever the synchronization service updates the clock parameters and estimated timing uncertainty of a given timeline reference, `qot_virtd` is responsible for conveying these changes back to the application, using the shared-memory region created by `ivshmem_server`.

Using the above components, we now describe their interactions which facilitate the transfer of QoT and timeline-related information between the applications deployed inside guest VMs and the services running on the host.

1) **Specifying QoT Requirements (Guest VM to Host):** To transfer application-specific QoT requirements from the guest to the host, we utilize the para-virtual VirtIO-serial interface, also referred to as `virtserial` [132]. VirtIO-serial provides bi-directional serial communication between applications running inside guest VMs with a host service. This interface is exposed to the guest application through a QEMU character-device driver front-end in the VM. Using an API, guest applications can read from or write messages to the character-device front-end. Since each VM runs as a QEMU process, the QEMU backend can forward guest application messages to a specified service on the host over a Unix socket. When an application in a VM binds to a timeline, the information is sent to `qot_virtd` using `virtserial` via the socket interface. The daemon then creates a version of the timeline on the host (using the QoT Core kernel module [4]), and registers the QoT requirements of the application with the host OS. `qot_virtd` also sends an acknowledgment to the guest application to indicate if the request was successfully accepted. Figure 4.3 highlights this interaction of each guest VM application with `qot_virtd`, and illustrates the transfer of application QoT requirements from a guest (VM 1) to the host using VirtIO-Serial. Although our stack supports multiple VMs, for the purpose of illustration, we show only one VM.

2) **Facilitating Low-Latency Clock Reads:** In QuartzV, we utilize the para-virtualized

dependent-clock paradigm and perform clock synchronization on the host on a per-timeline basis. Hence, we maintain a monotonic free-running core clock on the host, and compute its disciplining parameters (drift and offset), with respect to a global timeline reference. These disciplining parameters allow us to project the monotonic core clock to a global timeline reference. Therefore, to compute the current *timeline* time reference, a guest application needs to access a monotonic counter (on the host), and apply the clock-discipline parameters to this monotonic clock. Additionally, the synchronization service also computes the achieved QoT. This estimated QoT enables an application to read a timestamp with its associated uncertainty.

To enable low-latency reads of the timeline reference, we need to provide low-latency access to:

1. the monotonic core clock,
2. the timeline clock-projection parameters and,
3. the estimated QoT

We solve problem (1) by utilizing the para-virtual `KVM-clock`, which provides low-latency access to the host's real-time (`CLOCK_REALTIME`) and monotonic (`CLOCK_MONOTONIC`) clocks. Of these two clocks, `CLOCK_MONOTONIC` provides a monotonic clock source, and hence can be used as a core clock. Thus, `KVM-clock` allows the host OS and the guest VMs to, in practice, share the same core clock. Therefore, timeline clock-projection parameters calculated with respect to the host core clock can be applied (using Equation 1) inside the VM as well.

To solve problems (2) and (3), we use the inter-VM shared-memory (`ivshmem`) [133] interface to memory-map a shared-memory region containing the timeline clock parameters and uncertainty information into the guest VM application's virtual-memory space. Therefore, reading a timeline-reference timestamp involves reading `KVM-clock` and applying the timeline-projection and uncertainty parameters from shared memory. Since

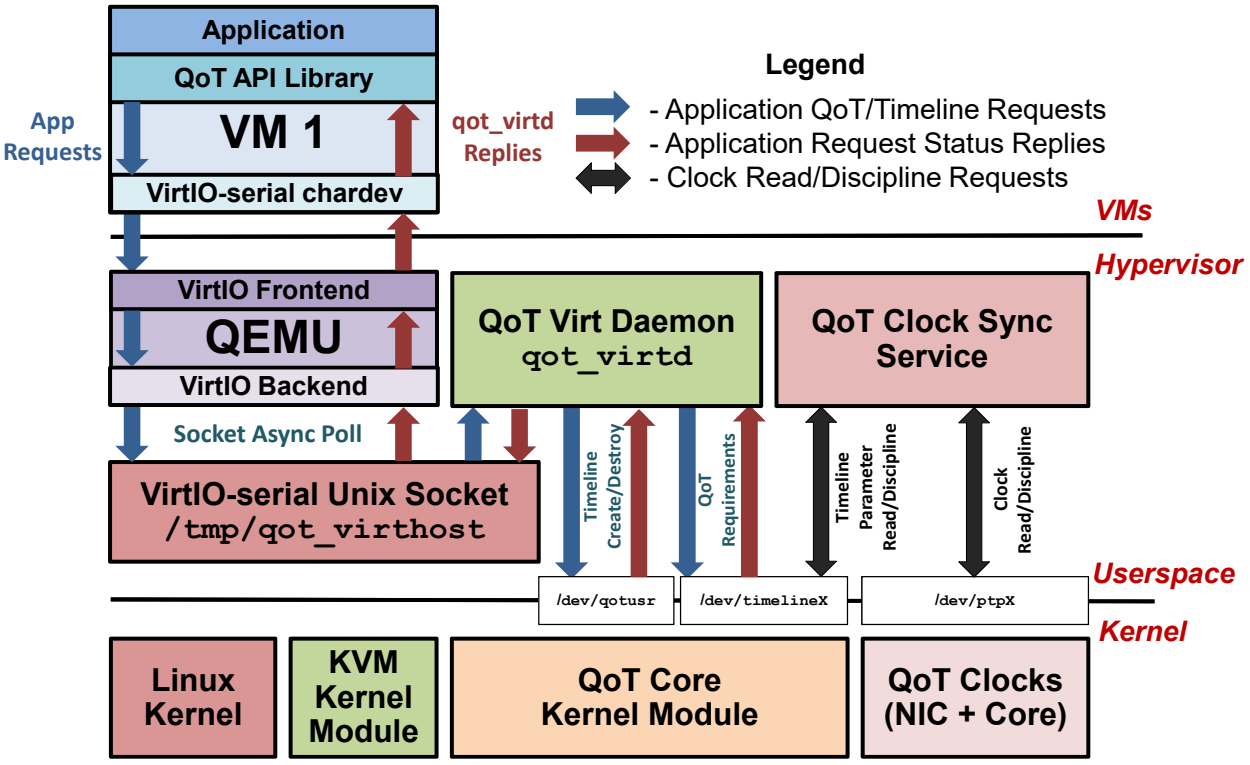


Figure 4.3: Specifying QoT information from guest applications to host service `qot_virtd` using VirtIO serial

these instructions are all unprivileged, the timeline-reference time can be read with low latency.

When a VM boots up, it registers with `ivshmem_server` over a Unix socket created by `ivshmem_server` (`/tmp/ivshmem_socket`). The server replies with a read-only file descriptor to the POSIX shared-memory region `/dev/shm/ivshmem` (created by `ivshmem_server`). `ivshmem` exposes this shared-memory region as a PCI device to the guest. When a guest application binds to a timeline, it interacts with this PCI device to memory-map the shared-memory region with read-only access into its own virtual-memory space. The fact that this shared-memory space is potentially shared across multiple VMs makes it necessary that VMs have read-only access. This provides isolation between different VMs while enabling low-latency clock reads.

In our implementation, we launch the `ivshmem_server` service on the host. This service provides a guest VM the right to access a read-only shared-memory region, which

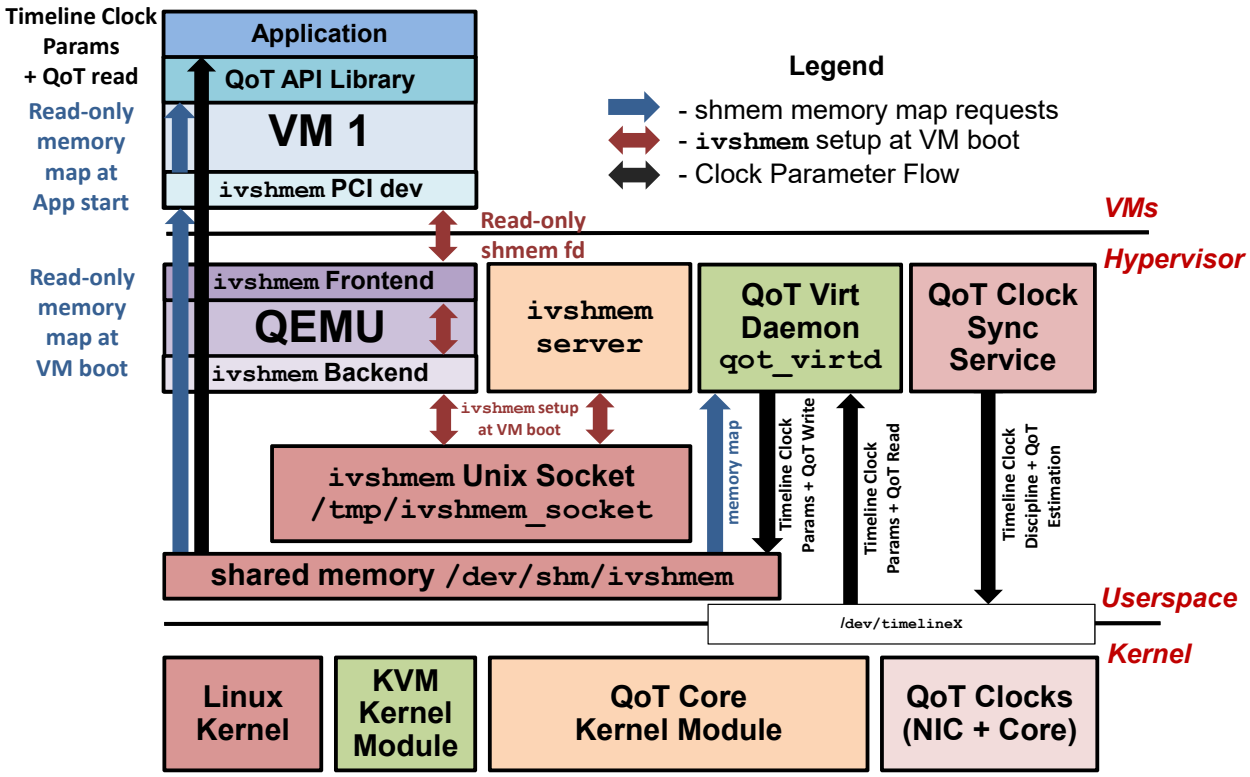


Figure 4.4: Sharing clock parameters and QoT information from host service `qot_virtd` to guest VM applications using `ivshmem`

contains the timeline clock parameters, and estimated uncertainty information. In addition to the guest VMs, the QoT Virtualization Service, `qot_virtd`, also memory-maps this shared-memory region with read-write access into its own virtual-memory space. Therefore, whenever the synchronization service updates the clock parameters and uncertainty information corresponding to a given timeline, `qot_virtd` writes these parameters to the shared-memory region which is memory-mapped into a guest application's virtual-memory space. Figure 4.4 highlights this interaction of guest VM applications with `ivshmem_server` and `qot_virtd`, and illustrates the sharing of per-timeline clock parameters and uncertainty information from host to guest (VM 1) using the memory-mapped shared-memory region created by `ivshmem_server`.

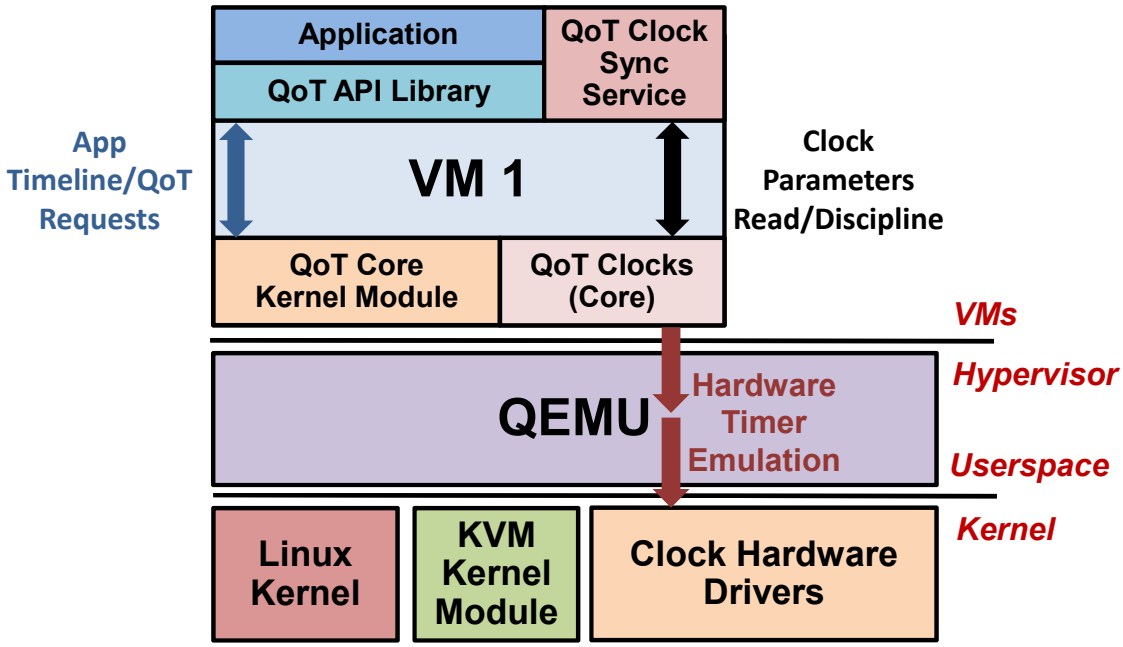


Figure 4.5: QoT Stack for Linux in a fully-virtualized guest VM

4.3.3 QoT and Full Virtualization

For guest VMs which do not support para-virtualized clocks, or hypervisors which do not permit extensions, the notion of QoT can still be supported. Our latest implementation of the QoT Stack for Linux allows all of its components: QoT core, QoT clocks, and clock-synchronization service (both NTP-based and PTP-based with software timestamping), to run inside a VM which does not support para-virtualization. However, the overhead of emulated hardware timers (full virtualization) will cause a loss in application performance, due to higher clock-read latency. Additionally, the overhead of an emulated network stack and lack of hardware-timestamping support (for PTP) can degrade the achieved synchronization accuracy and QoT. Figure 4.5 illustrates the components of the QoT Stack for Linux, deployed in a QEMU-KVM Linux VM (VM 1), which does not support para-virtualized clocks.

To support a core clock based on `CLOCK_MONOTONIC`, our latest implementation of the QoT Stack for Linux implements an architecture-independent QoT core clock driver [4], which allows the entire QoT stack to be deployed on any Linux-based platform includ-

ing VMs. This implementation provides a monotonic clock based on `CLOCK_MONOTONIC`, and provides time-based scheduling by using the existing Linux high-resolution timer (`HRTIMER`) interface.

4.3.4 QoT-based Industrial Automation using QuartzV

We now describe a simple test prototype to realize the industrial-automation application described in Section 4.2. We utilize the same structure as the described application and the main components are as follows:

1) **The Task Planner** running in a para-virtual VM with QuartzV is responsible for generating time-parametrized tasks. The VM is deployed atop QEMU-KVM on the desktop *Onyx* running Ubuntu 14.04 with a quad-core Intel i7 processor.

2) **Two Controller Nodes** each deployed on a Beaglebone Black [113] embedded platform (*Agate* and *Citrine*) with the QoT Stack for Linux, are responsible for generating and executing motion plans based on the time-based task plans.

3) **Simulated Robot Arms** receive motion plans from the controller nodes using the ROS-based [134] publish-subscribe mechanism. Since we did not have ready access to real robots, we utilize ROS Indigo [134] with the Gazebo simulator [135] to simulate two Universal Robotics UR5 [130] robot arms along with their motion controllers (using `ros-control` [136]). The simulation is performed on the desktop machine *Jasper* running Ubuntu 14.04 with a quad-core Intel i7 processor and an Nvidia GT620M GPU.

We consider a simple scenario where two robots collaboratively pick and place a component synchronously. However, our testbench can be used to develop and test more complex application scenarios. Additionally, the use of ROS enables the same application code to be deployed directly on a real robot. A video showing our prototype application can be found at `https://youtu.be/7NoxnZEWDrM`.

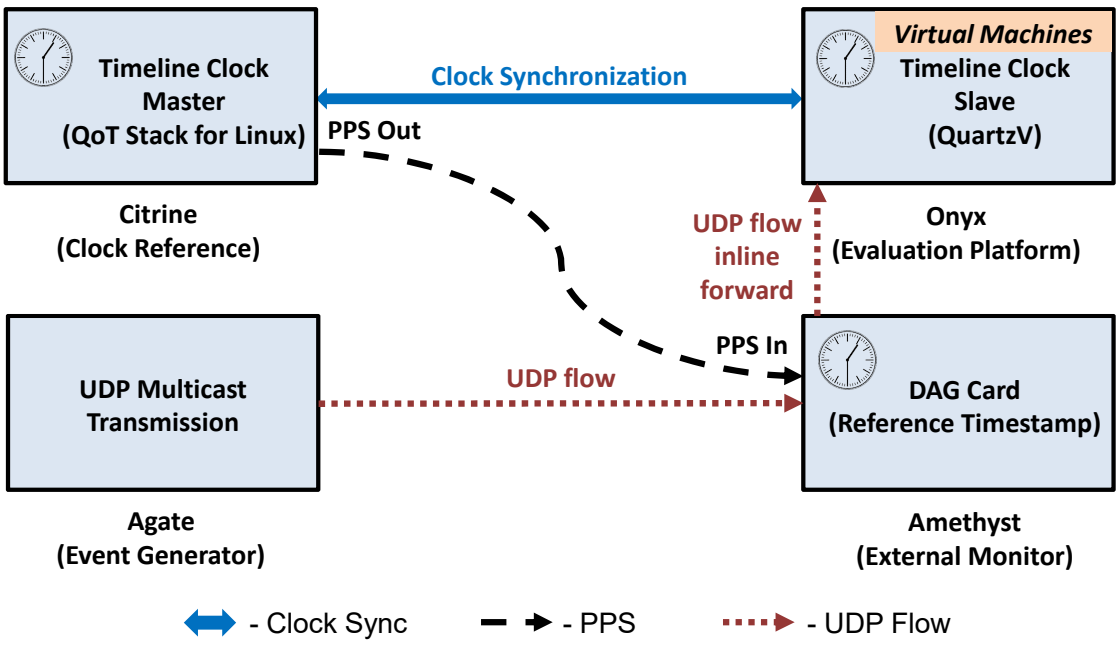


Figure 4.6: QuartzV Clock-Synchronization Test-bed

4.4 Experimental Evaluation

We now present some experimental results to benchmark the performance of QuartzV using as metrics (i) clock-synchronization accuracy, and (ii) clock-read latencies. We use the QoT Stack for Linux deployed natively as the baseline. Before stating the results, we describe our experimental setup.

4.4.1 Experimental Setup

Figure 4.6 illustrates the different nodes in our clock-synchronization test-bed. All the nodes are interconnected by an IEEE 1588 (PTP)-compliant Ethernet switch [137].

Our evaluations are performed on a quad-core (8 virtual cores) x86-64 Intel i7-based desktop *Onyx*, which hosts the QoT-based benchmarking applications. *Onyx* utilizes Ubuntu 14.04 with the Linux 4.4 kernel and also contains version 2.8 of the QEMU-KVM hypervisor. This enables *Onyx* to host VMs utilizing QuartzV. The Intel i7 CPU contains a constant-invariant TSC which always maintains a steady frequency, and thus can be used as a reliable clocksource. Additionally, *Onyx* is equipped with an IEEE 1588 (PTP)-

compliant Intel 82574L network interface [138] which supports hardware timestamping at the PHY layer. The presence of hardware timestamping allows us to perform accurate clock synchronization using the QoT Stack for Linux's PTP-based synchronization service.

We utilize the Beaglebone Black node *Citrine* as our clock reference. The Beaglebone Black ARM-based TI AM335x chipset [113] contains an IEEE 1588-compliant network interface which supports hardware timestamping at the PHY Layer.

To measure the accuracy of clock synchronization on *Onyx*, with respect to the reference node *Citrine*, we utilize the nodes *Amethyst* and *Agate*. To measure synchronization accuracy, we need to take (near) simultaneous timestamps of a common event on both the reference (master) and the target (slave). By comparing these timestamps over a period of time, we can compute the synchronization accuracy. Therefore, we use (i) the node *Agate* (Beaglebone Black) to periodically (every second) generate UDP-multicast packets which serve as common-reference events providing timestamping opportunities, and (ii) the node *Amethyst* to generate reference timestamps (equivalent to *Citrine*) for the UDP datagrams.

Amethyst has an x86-64 Intel i7 processor, running Ubuntu 14.04 with the Linux 4.12 kernel, and is equipped with an Endace 7.5G2 DAG card [139]. The DAG card contains two ports which intercept all packets flowing between *Agate* and *Onyx*. This card also provides 7.5 nanosecond resolution timestamping [139], and processing of packets at line rate. Therefore, all the UDP packets from *Agate* can be accurately timestamped with no significant delay introduced by the DAG card. The same UDP packets can subsequently be timestamped on *Onyx* (using socket/hardware timestamping [140] on the host and guest VMs). Hence, if we assume (for now) that the DAG card on *Amethyst* can provide equivalent timestamps as the reference *Citrine*, then by comparing these timestamps with those (nearly) simultaneously generated on *Onyx*, we can compute the clock-synchronization accuracy of *Onyx* with respect to *Citrine*. Note that the introduction of the DAG card adds noise to our measurements, as there is some latency between

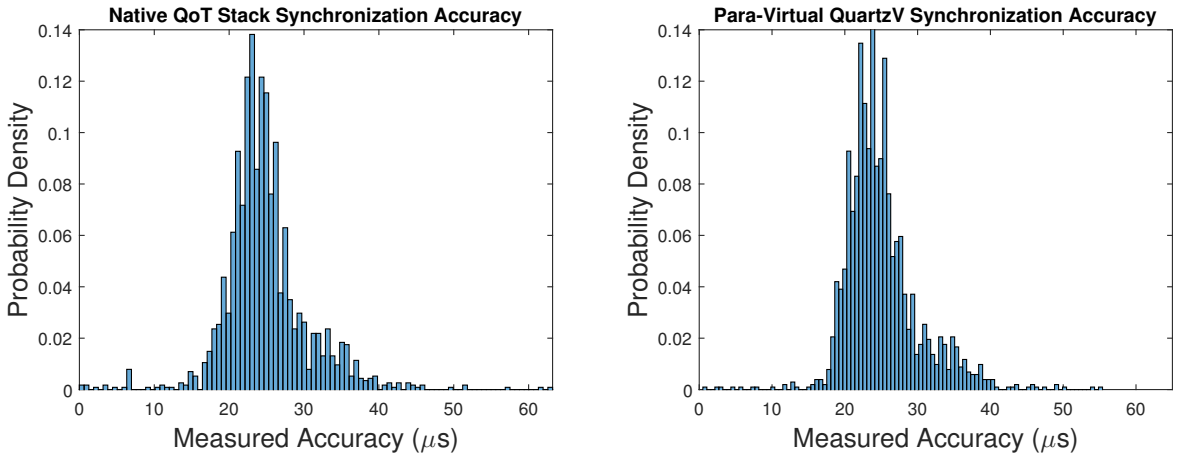
the DAG timestamp and the timestamp on *Onyx*. However, given that *Onyx* and the DAG card share a dedicated link, the latency is low.

To accurately synchronize the DAG card on *Amethyst* to the reference *Citrine*, we utilize its inbuilt PPS (Pulse-per second) input. We use *Citrine* (Beaglebone Black) to generate a reference PPS signal over a GPIO pin (using a hardware timer), which is fed to the PPS input of the DAG card. The DAG card can use this signal along with *Amethyst's* system clock (`CLOCK_REALTIME`) to precisely synchronize its clock with <10ns accuracy. Hence, to achieve precise synchronization (using PPS), we also need to synchronize *Amethyst's* system clock (`CLOCK_REALTIME`) to the reference clock on *Citrine*, with an accuracy within 1s. This can be done using Linux PTP's [141] two-stage system-clock synchronization (*ptp4l* and *phc2sys*). *Amethyst* is also equipped with an IEEE 1588-compliant Intel 82574L network interface [138] which supports hardware timestamping at the PHY layer. Hence, using PTP, we can synchronize `CLOCK_REALTIME` to the reference clock on *Citrine* to an accuracy on the order of microseconds, which is more than sufficient compared to the requirement of within 1s. Along with PPS, this allows us to achieve DAG clock synchronization with accuracy on the order of a few nanoseconds. Therefore, we can externally measure the synchronization accuracy of QuartzV.

4.4.2 Synchronization Accuracy

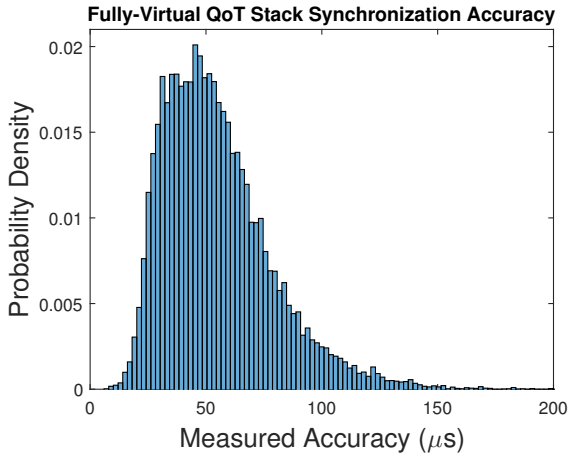
We now compare the clock-synchronization accuracy (or error), with respect to the reference *Citrine*, achieved by (i) QuartzV for a Linux VM with para-virtual-clock support, (ii) the QoT Stack for Linux deployed natively, and (iii) the QoT Stack for Linux deployed in a VM with a fully-virtualized clock. Note that, in cases (i) and (ii), clock synchronization happens on the host OS, while, in case (iii), clock synchronization happens inside the VM. We use Ubuntu 14.04 VMs, each configured to use 2 Virtual CPU cores and 2 GB of memory.

Figure 4.7 shows the histogram of the measured clock-synchronization accuracy, and



(a) Native

(b) Para-virtual QuartzV



(c) Full virtualization

Figure 4.7: Measured Clock-Synchronization Error Distributions. The y-axis represents the probability density, and the x-axis the measured error

Figure 4.8 shows a box-plot of the clock-synchronization accuracy for the mentioned scenarios. The measurements were taken over a period of six hours. Notice that the accuracy distribution achieved by the QoT Stack natively (Figure 4.7a) and QuartzV for para-virtual VMs (Figure 4.7b) is nearly identical with a mean of 24.28μs and 26.12μs respectively, and standard deviation of 5.05μs and 5.12μs respectively. This is because QuartzV performs clock synchronization on the host and transfers the clock-projection parameters to the guest VM. On the other hand, the accuracy achieved by the fully-virtualized QoT Stack inside a VM (Figure 4.7c) is lower with a mean of 70.23μs and a standard deviation of 128.28μs. This is due to the additional packet-timestamping

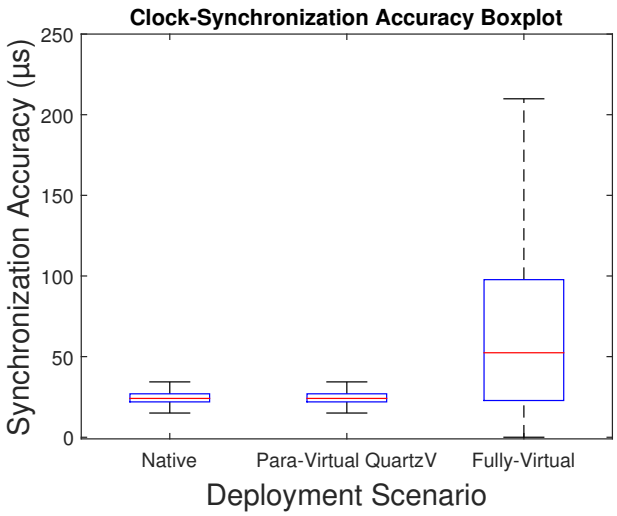


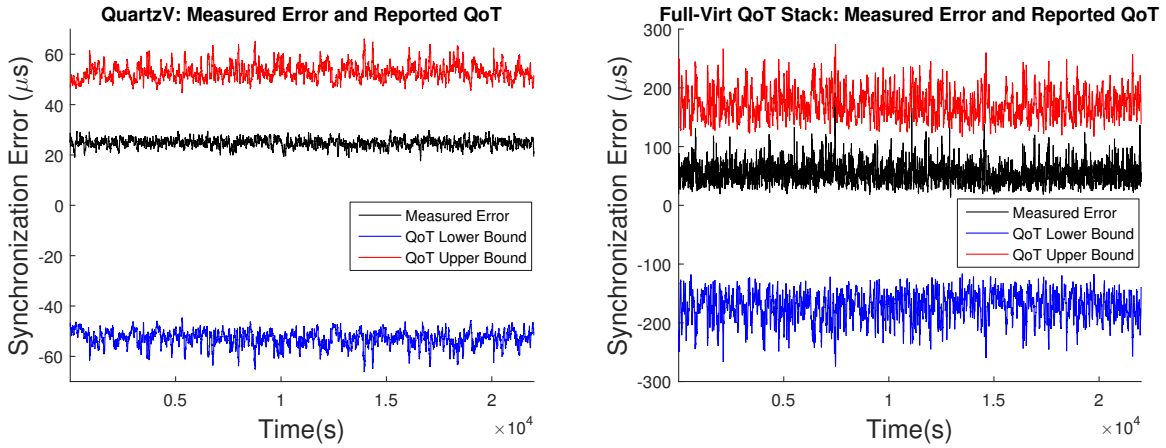
Figure 4.8: Clock-Synchronization Accuracy Boxplot. The center 'red' line represents the median accuracy, the inner whiskers the 25th and 75th percentile accuracy, and the outer whiskers the minimum and maximum error observed.

uncertainty introduced by the virtualized networking stack.

The ability to provide QoT bounds also enables fault detection. Figures 4.9a and 4.9b plot the upper and lower QoT bounds calculated by para-virtual QuartzV, and the fully-virtualized QoT Stack deployed inside a VM respectively. Observe that the computed bounds always bound the accuracy measured by the experimental test-bench. From these results, we can conclude that QuartzV can provide near-native clock-synchronization accuracy to applications running in VMs.

4.4.3 Clock-Read Latency

To compare clock-read latencies, we consider the following scenarios: (i) the QoT Stack for Linux deployed natively, with the x86 Time-Stamp Counter (TSC) clocksource, (ii) a para-virtual Linux VM using QuartzV with the `KVM-clock` [57] clocksource, and (iii) the QoT Stack for Linux deployed in a VM with an emulated (fully-virtualized) x86 High-Precision Event Timer (HPET) clocksource. For each of these cases, we measure the latency of reading a timeline reference, which is calculated by applying the projection parameters to the QoT core clock, `QOT_CORE` (based on `CLOCK_MONOTONIC`). To measure a clock's read latency, we read the clock in a continuous loop, and take the difference be-



(a) Para-virtual QuartzV (b) Fully virtual QoT Stack

Figure 4.9: QoT Bounds: (a) para-virtual QuartzV, (b) fully-virtual QoT Stack

tween adjacent readings. For the sake of comparison, we also present latency measurements for the clocks exposed by Linux (`CLOCK_MONOTONIC` and `CLOCK_REALTIME`) along with the x86 TSC.

The clock-read latency data can be found in Table 4.1. We present the minimum, average and standard deviation of the latency measurements for all of the clocks being compared. The data is averaged across 1000 experiments, each consisting of 1 million consecutive clock reads. Observe that, for `CLOCK_MONOTONIC`, `CLOCK_REALTIME` and `QOT_CORE`, the average and minimum clock-read latency observed in the para-virtual guest VM is roughly twice (~2x) that observed in the native environment. This reflects the overhead introduced by using the para-virtual `KVM-clock` as a clocksource. Compare this with the fully-virtualized case which has latencies that are 3 orders of magnitude (>100x) greater than the native setting. This is due to the overhead of emulating the HPET clocksource. On the other hand, reading the TSC (using the `rdtsc` instruction) has nearly the same latency in all three scenarios. This is because `rdtsc` is an unprivileged instruction and can be executed natively [121].

`QOT_CORE` (QoT core clock) is implemented as a wrapper around `CLOCK_MONOTONIC`. Observe that, in all the three cases, the observed latency in reading `QOT_CORE` is slightly greater than `CLOCK_MONOTONIC`. This is because of the additional overhead of applying the

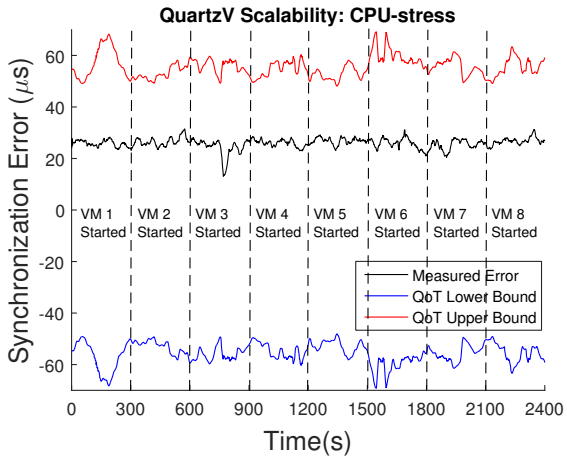
Table 4.1: Clock-Read Latency (nanoseconds)

Scenario	Clock	Min	Average	Std. Dev
Native QoT Stack (x86 TSC)	TSC	4	7.41	59.55
	REALTIME	13	26.19	172.44
	MONOTONIC	13	18.41	95.74
	QOT_CORE	16	32.01	123.69
Para-virtual QuartzV (KVM-clock)	TSC	4	8.28	88.75
	REALTIME	31	40.46	246.47
	MONOTONIC	31	34.79	233.83
	QOT_CORE	54	60.71	242.34
Fully-virtual QoT Stack (Emulated HPET)	TSC	4	8.19	95.18
	REALTIME	1785	2038.02	9721.72
	MONOTONIC	1786	2022.13	8912.25
	QOT_CORE	1892	2435.64	9512.45

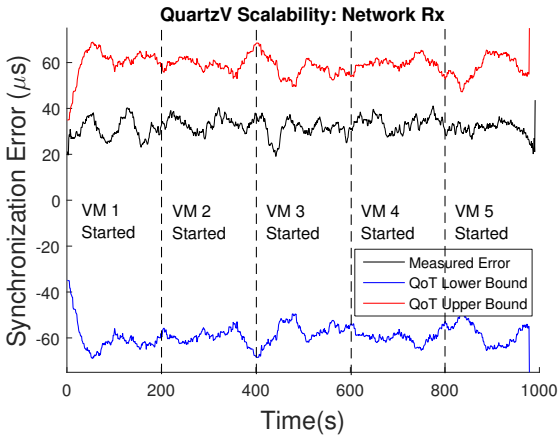
timeline clock-projection parameters. For the para-virtual scenario using QuartzV, the QOT_CORE latency is ~1.8x that of CLOCK_MONOTONIC. This is due to the overhead of accessing the shared-memory region exposed by ivshmem. However, this overhead is minimal and does not affect the order of magnitude of the clock-read latency, as compared to CLOCK_MONOTONIC.

If we compare the two virtualization scenarios based on standard deviation, we can observe that reading the para-virtual KVM-clock clocksource provides approximately 40x lower standard deviation (clock-read latency variability) than an emulated clocksource. This lower variability translates to better QoT. Additionally, note that the QuartzV implementation of the QOT_CORE clock has similar standard deviation as CLOCK_MONOTONIC. Therefore, we conclude that QuartzV provides minimal loss in timing performance (latency and uncertainty) compared to the native case, while allowing services on the host to expose the notion of Quality of Time to applications running in guest VMs.

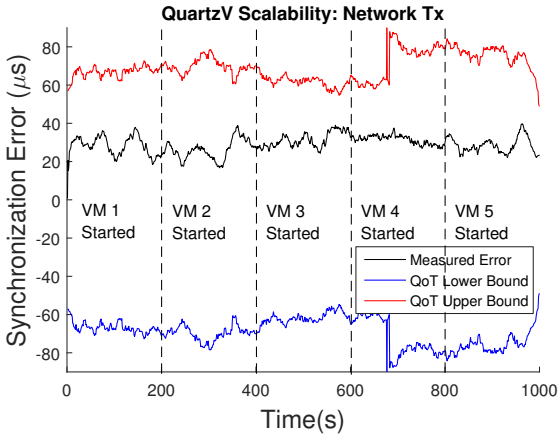
Notice that for both the para-virtual and fully-virtual scenarios, the clock-synchronization error is an order of magnitude (>10x) higher than the clock-read latency. Thus, the network residency and timestamping uncertainties are the bottlenecks for achieving good QoT for an application in a VM.



(a) CPU Stress



(b) Network Rx



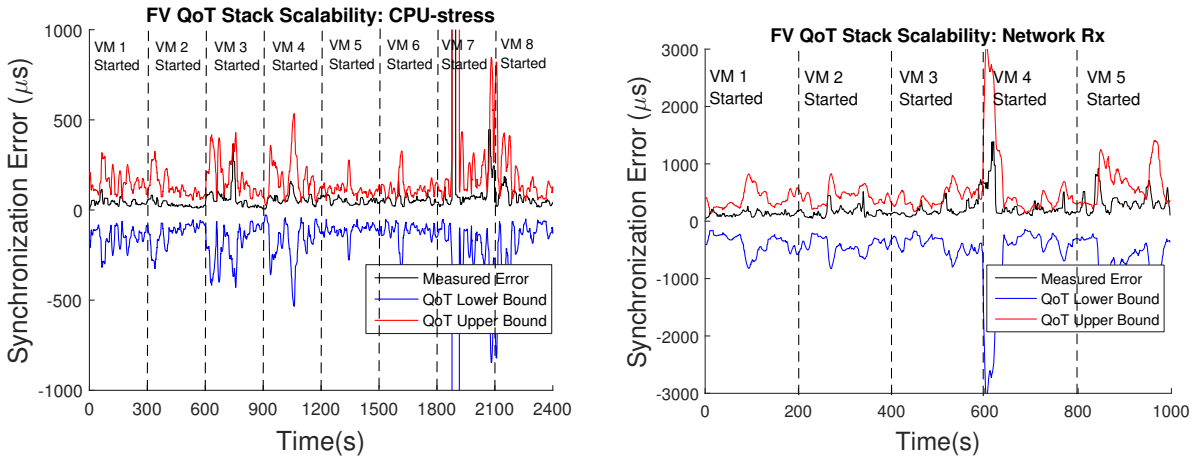
(c) Network Tx

Figure 4.10: QuartzV Synchronization Scalability Results. The dashed lines represent moments in time where a new VM was spawned

4.4.4 Clock-Synchronization Scalability

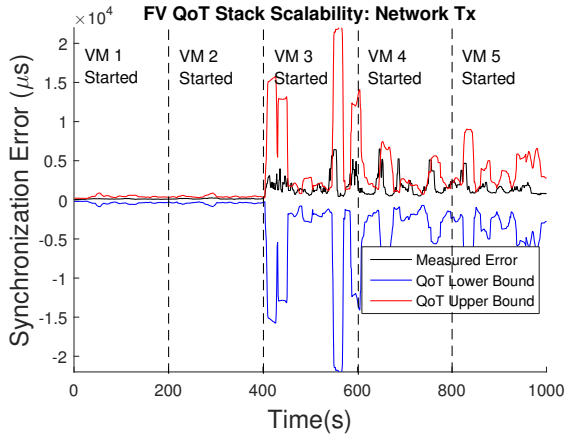
We now analyze the scalability of (i) QuartzV for Linux VMs with para-virtual clock support, and (ii) the QoT Stack for Linux deployed in a VM utilizing full virtualization. Our experiments measure the clock-synchronization accuracy achieved in the presence of competing VMs present on the same host. To test the limits of both approaches, we consider scenarios involving competing VMs with CPU and network-intensive workloads.

Figures 4.10 and 4.11 provide the scalability results for the para-virtual QuartzV setup, and the fully-virtual QoT Stack respectively. In both figures, subplots (a) provide



(a) CPU Stress

(b) Network Rx



(c) Network Tx

Figure 4.11: Fully-Virtual QoT Stack Synchronization Scalability Results. The dashed lines represent moments in time where a new VM was spawned

results in the presence of competing VMs with CPU-intensive workload, subplots (b) provide results in the presence of competing VMs with network data-reception-intensive workload, and subplots (c) provide results in the presence of competing VMs with network data-transmission-intensive workload. Each plot shows the measured clock-synchronization accuracy and reported QoT bounds. The x-axis denotes the progression of time in seconds, and the y-axis indicates the measured synchronization error in microseconds. Please note that each sub-plot has a different scale for the y-axis.

Figures 4.10a and 4.11a present scalability results when multiple VMs with CPU-intensive workload are present. To test the limits of our approach, we consider a maxi-

mum of 8 VMs, each with 1 virtual core and 2 GB of memory, as our test-node *Onyx* has 8 virtual cores and 16 GB of memory. Each VM runs a simple QoT-aware application, which binds to a timeline, and reads the timeline clock in a tight loop with real-time priority. In addition, each VM also utilizes the *stress* tool [142] to spawn a single CPU-intensive thread, without real-time priority. This ensures that any CPU capacity left over by the QoT-aware application will be consumed by the stress tool. We spawn a new VM every 300 seconds, and the dashed lines in the plot represent points in time where a new VM was spawned. In practice, we observe that our test-bed system's CPU is fully utilized with 6 CPU-intensive VMs. This is due to the use of some processing capacity by the host OS, the graphics sub-system, and QEMU-KVM.

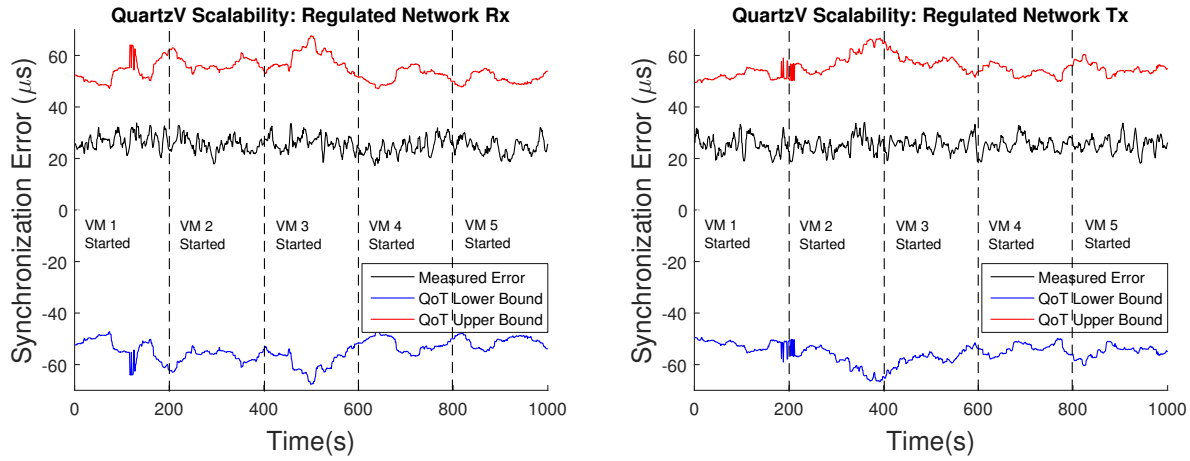
Observe that, for the para-virtual QuartzV case (Figure 4.10a), there is no significant change in synchronization accuracy as new CPU-intensive VMs are spawned. This is because clock-synchronization is performed in the host OS, and as long as the synchronization service has sufficient resources, the accuracy remains unaffected. Additionally, the use of hardware timestamping (available only on the host), ensures that the packet-timestamping uncertainty is unaffected by CPU load. On the other hand, for the fully-virtualized QoT Stack (Figure 4.11a), as the VM count grows higher, the synchronization accuracy degrades, and greater instability can be observed in the obtained accuracy. This is because clock synchronization is performed inside the VM, and the networking stack is emulated by the hypervisor. Thus, greater CPU load increases the uncertainty in the software timestamping of synchronization packets, and makes the synchronization service unstable. This in turn degrades accuracy. Specifically, after the addition of the 7th VM, the system is overloaded, and there are durations where the synchronization accuracy is significantly degraded ($>$5 times the case without overload). The QoT bounds returned by the system reflect this instability in the fully-virtual synchronization service.

Figures 4.10b, 4.10c, 4.11b and 4.11c present scalability results when multiple VMs with network-intensive workloads are present. In these experiments, we consider a single VM running a QoT-aware application, and a maximum of 5 competing VMs, each

with 1 virtual core and 2 GB of memory, and no per-VM bandwidth restrictions. We spawn a new VM every 200 seconds, and the dashed lines in the plot represent points in time where a new VM was spawned. Each VM uses the *iperf* tool [143] to send/receive TCP packets to/from another machine on the LAN, such that the available network bandwidth is saturated. We observed that, without bandwidth regulation, a single VM is able to nearly saturate the network bandwidth. Further adding new VMs causes the load to grow incrementally until VM 4, after which the bandwidth is fully saturated. This is because, in our setup, the 100 Mbps industrial PTP switch [137] is the network bottleneck, as compared to the 1 Gbps Ethernet card on the host *Onyx*.

Notice that, for the para-virtual QuartzV case, with network data-reception-intensive workload (Figure 4.10b), the achieved synchronization accuracy and uncertainty (variance) degrades by ~1.2x, as compared to the load-free scenario shown in Figure 4.9. However, this degradation is minimal and does not significantly change as new competing VMs are added. This is because clock synchronization is performed on the host, and uses hardware timestamping. Similarly, for the para-virtual QuartzV case with network data-transmission-intensive competing workload (Figure 4.10c), the achieved synchronization accuracy is similar to the network data-reception-intensive case. However, the synchronization uncertainty (variance) degradation is higher by ~1.3x, as compared to the previous case. This observation especially holds true when more competing VMs are present (> 3), and is reflected by the increase in the reported QoT bounds.

On the other hand, for the fully-virtualized QoT Stack (Figures 4.11b and 4.11c), as the competing network-intensive VMs increase, the synchronization accuracy degrades significantly on average (~1.8x-4x in different regions). Moreover, at the instances where new VMs are added, greater instability can be observed in the obtained accuracy. Also, observe that, for the network data-reception-intensive case, the accuracy significantly degrades on the addition of the fourth VM, and for the data-transmission intensive case, this can be observed at the point of addition of the third VM. The accuracy degradation is one order-of-magnitude worse for the network data-transmission-intensive case, and



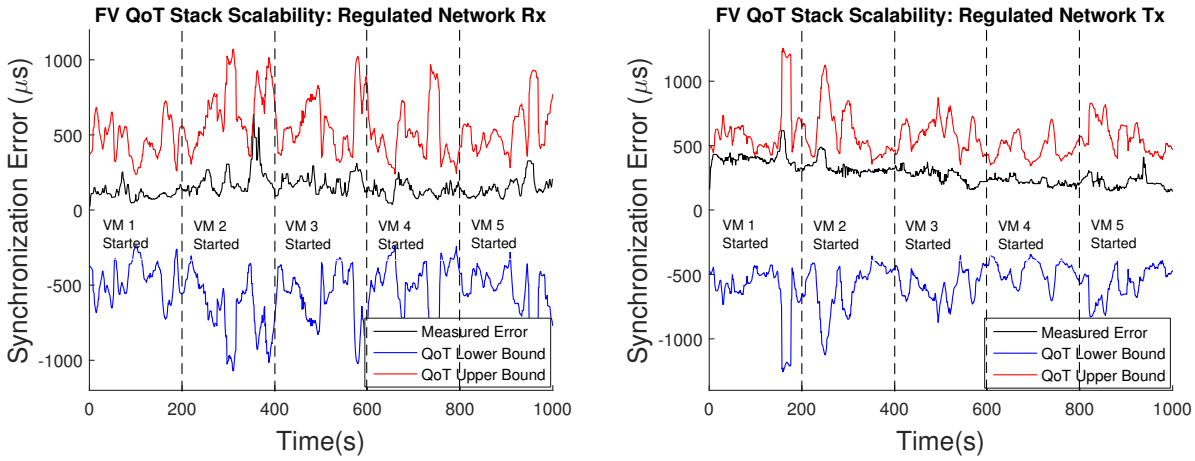
(a) Regulated Network Rx (b) Regulated Network Tx

Figure 4.12: QuartzV Synchronization Scalability Results with per-VM Network Reception/Transmission Bandwidth restricted to 2 MB/s

this is reflected by the QoT bounds returned by the system, which are, in the worst case, about ~10x of those reported in the presence of network data-reception-intensive load.

For both the para-virtual and fully-virtual scenarios, the accuracy degradation observed is greater in the presence of data-transmission-intensive network load. This is because, for the network-reception case, as the incoming traffic increases, there is more congestion at the PTP switch, as the switch is the bottleneck. On the other hand, for the network-transmission case, as the switch becomes congested, packets start getting dropped, and there are more re-transmission attempts made at the host Ethernet card (due to TCP), thus causing greater congestion at the host. However, the degradation of both the measured accuracy and computed QoT bounds observed while using para-virtual QuartzV is minimal, as compared to the significant degradation observed while using the fully-virtualized QoT Stack inside a VM. This is explained by the fact that, during overload, the overhead of using an emulated networking stack creates greater uncertainties and delays in handling and timestamping synchronization packets.

In summary, our scalability experiments indicate that, for the para-virtual QuartzV approach, CPU-intensive VMs do not significantly affect clock-synchronization accuracy when: (i) adequate hard CPU reservations are used (already guaranteed by default in all



(a) Regulated Network Rx (b) Regulated Network Tx

Figure 4.13: Fully-Virtual QoT Stack Synchronization Scalability Results with per-VM Network Reception/Transmission Bandwidth restricted to 2MB/s

hypervisors), (ii) Virtual Machine over-commit is avoided (i.e., not allowing more VMs than available resources), and (iii) by ensuring that the clock-synchronization service has sufficient resources. However, the same cannot be said for the fully-virtualized QoT Stack deployed inside a VM. For the network-intensive scalability experiments, we have observed that, for both QuartzV and the fully-virtual QoT Stack, a heavy network load does affect the clock-synchronization accuracy and the reported QoT bounds. The degradation in the observed accuracy is significant for the fully-virtual QoT Stack while being minor for the para-virtual QuartzV approach. This degradation is caused due to added uncertainty in network timestamping and packet residency delays, and can be avoided by restricting the network bandwidth available to a VM, based on a user-specified limit. Such functionality is available in most hypervisors including QEMU-KVM.

Figures 4.12 and 4.13 present scalability results in the presence of bandwidth-restricted network-intensive VMs, for para-virtual QuartzV and the fully-virtual QoT Stack respectively. We consider a maximum of 5 competing VMs, each of which has its transmission and reception bandwidth restricted to 2 MB/s (16 Mbps). In both figures, subplots (a) provide results in the presence of competing VMs with network data-

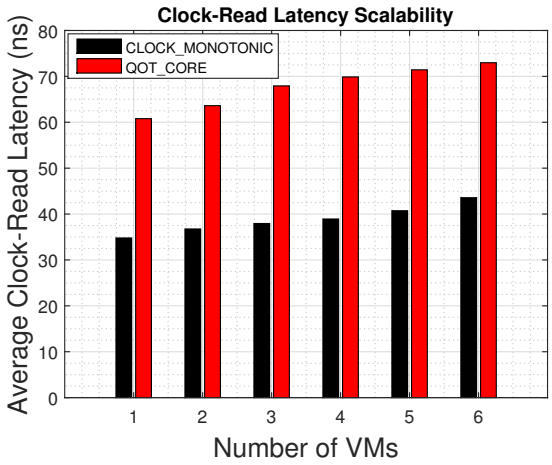


Figure 4.14: QuartzV Clock-Read Scalability Results

reception-intensive workload, and subplots (b) provide results in the presence of competing VMs with network data-transmission-intensive workload. For both the para-virtual QuartzV scenario and the fully-virtual QoT Stack, the plots indicate that restricting the bandwidth of competing VMs can prevent significant degradation of synchronization accuracy, as compared to the scenario with no bandwidth restrictions.

4.4.5 Clock-Read Scalability

Figure 4.14 plots the average clock-read latency of the para-virtual QuartzV approach with multiple VMs continuously performing simultaneous clock reads. Observe that for both `QOT_CORE` and `CLOCK_MONOTONIC`, the clock-read latency increases slightly for each new VM spawned. This is due to the unavoidable contention in reading the hardware counter to compute the time. Thus, as `qot_virtd` writes the clock-discipline parameters to a shared-memory region which all VMs can simultaneously read from, there is no bottleneck in our implementation, allowing QuartzV to easily scale and support multiple VMs.

4.5 Summary

Given that virtualization is increasingly utilized in cyber-physical applications, we introduced the QuartzV extension to our QoT Stack for Linux to make virtual machines (VMs) QoT-aware. QuartzV harnesses para-virtual clocks along with the dependent-clock paradigm [25] to provide near-native timing performance in VMs. We also demonstrated the utility of QuartzV by using it in a prototype industrial-automation application. This, in turn, illustrates that QoT-awareness makes it possible for intelligent CPS applications to dynamically take coordination decisions, based on a shared notion of time and the delivered QoT.

For VMs which do not support para-virtual clocks, or hypervisors which do not permit extensions, we extended the QoT Stack for Linux so that it can be entirely deployed in a VM. However, our experiments indicate that QuartzV's para-virtual implementation can achieve much higher synchronization accuracy, better scalability and timing performance. QuartzV is open-source, and the code can be found at https://bitbucket.org/rose-line/qot-stack/src.

Chapter 5

Time-as-a-Service for Geo-distributed Coordination

Modern distributed applications are inherently complex and consist of multiple interacting components. Thus, deploying these components and managing their life-cycles are complicated endeavors. Additionally, many of these components will be deployed in the cloud or at the edge in conjunction with other applications. In such scenarios, the use of OS-level virtualization technologies like containerization [27] simplifies the deployment and life-cycle management of distributed applications. Therefore, Quartz builds on the QoT Architecture [4] for providing *Time-as-a-Service* (TaaS) to containerized applications. Quartz features a distributed modular architecture and is implemented using containerized micro-services, making it easy to deploy and use across a range of platforms.

Unlike the kernel-space QoT Stack [4] which operated at LAN-scale, Quartz overcomes the scalability and portability issues by featuring a fully user-space implementation which (i) supports multi-tenancy, (ii) operates at geo-distributed (WAN)-scale, and (iii) is portable to an array of application domains and platforms. Quartz also provides an API for distributed coordination based on the *timeline* abstraction [4], and allows distributed application components to specify their required QoT. Based on these requirements, Quartz orchestrates the underlying system and clock-synchronization protocols

to meet these application-specific requirements, and feeds back the delivered QoT back to the application.

The key contributions described in this chapter are as follows:

1. Elucidating the challenges and subsequent architectural choices in exposing Time-as-a-Service, maintaining timelines and estimating Quality of Time (QoT) at geo-distributed scale,

2. Introducing techniques to make clock-synchronization protocols, adaptive to application QoT requirements, and

3. Introducing Quartz, an autonomous, adaptive and fault-tolerant middleware exposing Time-as-a-Service for containerized applications using time as a coordination primitive.

5.1 An Application's Perception of Time

We first motivate the utility of Quartz by describing two application scenarios which can be enabled by using a shared notion of time and QoT. These applications are (i) *DronePorter*, a fleet of drones coordinating to transport a payload, and (ii) *TimeCop*, a traffic-management solution which coordinates vehicular traffic flow at city scale in both space and time. However, the core concepts can be adapted to other distributed-coordination application domains.

DronePorter: Consider a fleet of n drones (as shown in Figure 1.2 in Chapter 1) transporting an object Ω, too large to be carried by a single drone. To successfully transport Ω, the drones need to follow a coordinated flight-plan such that (i) the object is not damaged or destabilized, and (ii) the drones do not collide with each other or obstacles in the environment. One way to accomplish this is by having a master entity, which can be one of the drones, send out timestamped flight-plans with way-points to

each of the drones, such that each drone tries to reach a given way-point at the specified time.

To coordinate successfully, the clock on each drone needs to be synchronized such that the accuracy is within some specified limits. This accuracy (or uncertainty) specification can depend on multiple factors, ranging from the velocity and size of the drones, to the other uncertainties in the environment. For example, to meet a particular velocity, while maintaining safety, having a tighter clock-synchronization accuracy can be used to compensate for higher localization uncertainties or higher environmental uncertainties [18]. Therefore, in this scenario, each drone can use Quartz to bind to a timeline each specifying its QoT requirements. If the QoT deviates beyond these requirements, the drones can be notified, and can adapt by moving into a graceful-degradation mode. Additionally, as shown in Figure 1.2, we can also have an edge/cloud controller also join the timeline, and provide (i) high-level objectives/guidance to the fleet of coordinating drones, and (ii) fleet-management capabilities. Note that such a cloud/edge controller can provide a higher level of macroscopic control at a lower frequency, and hence can have less-stringent QoT requirements than the drones.

TimeCop: Consider a city with an adaptive traffic signal deployed at each intersection, which contains: (i) a traffic signal with an interface through which the phase (traffic-signal state) can be set, and (ii) camera-based sensors which provide per-lane queue lengths (number of vehicles) at the intersection.

Each intersection is controlled by a traffic controller, which can be deployed on an edge device at or near the intersection, for low-latency decision-making. This controller is responsible for controlling the timing and phase of the traffic signals at the intersection. The traffic controller makes decisions periodically, by taking as input (i) the number of vehicles per ingress lane at the intersection (read from the traffic sensors) in the last interval, and (ii) the number of vehicles inbound from adjacent intersections (published by the adjacent intersections). The generated output is the next phase of the traffic signal. In this scenario, a shared notion of time is key to ensure that (i) the state from

adjacent intersections has accurate timestamps, and (ii) the phase of the traffic signals at an intersection can be switched at an accurate time instant to ensure efficient traffic flow. Thus, each intersection controller uses Quartz to bind to the *traffic-management* timeline with a QoT requirement of +/-1 ms, while Quartz ensures that all controllers bound to the timeline share the same notion of time with the desired QoT specification. Thus, the timeline abstraction allows a coordinating group of endpoints to be specified. Quartz also ensures that every timestamp is appended with accurate QoT estimates, enabling controllers to decide "data validity" based on the QoT bounds, i.e., data with QoT bounds beyond tolerable limits can be discarded or used with abundant caution. Figure 5.1 illustrates the TimeCop solution.

Consider a scenario where multiple applications such as *TimeCop* and *DronePorter* are deployed on the same infrastructure. For example, DronePorter's high-level controller can be deployed on the same edge device as TimeCop's per-intersection traffic controllers. One can also envision a situation where multiple such emerging smart-city applications are deployed on the same infrastructure. In such a scenario, the ability to simultaneously maintain multiple per-application timelines allows (i) each application's coordinating components and their QoT requirements to be individually specified, and (ii) allows the system to meet potentially different QoT requirements of each application.

With each application component specifying the required QoT, the system knows the maximum level of uncertainty tolerable by the distributed-coordination application. Since each node independently computes its QoT with respect to the reference, a node can enter a graceful-degradation [126] mode when the level of uncertainty exceeds the tolerable limit. Additionally, if a coordination message is delayed or arrives too late, all a node needs to do is compare the message timestamp against the current time on its local clock [16]. Also, given that commodity oscillators drift slowly, the probability of clock-synchronization failure is much lower than the probability of CPUs, networks or disks failing [6]. Therefore, utilizing a shared notion of time with the added notion of QoT can enable scalable and fault-tolerant coordination [17].

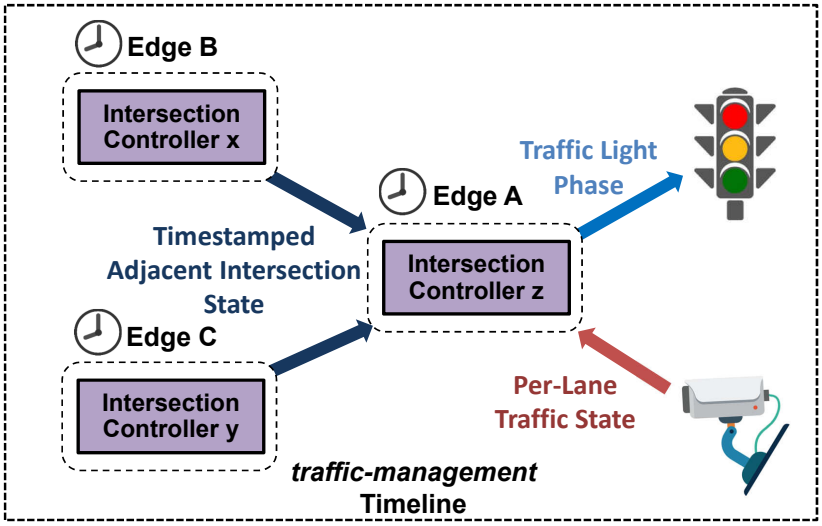


Figure 5.1: *TimeCop*: City-Scale Traffic Management

5.2 Quartz: *Time-as-a-Service* (TaaS)

We now introduce Quartz which exposes *Time-as-a-Service* to containerized applications. We describe Quartz by starting at the application level and then explaining the high-level capabilities Quartz provides through its API. Subsequently, we focus on its architecture, design choices and its implementation as micro services.

5.2.1 Quartz: API

Quartz features a rich application-programming interface (API) that is centered around the notion of a *timeline* – a virtual sense of time to which applications bind with their desired accuracy level and minimum clock resolution [4]. A timeline is the key primitive specifying the application components which coordinate with each other. The Quartz API provides applications the ability to (i) bind/unbind from a timeline, (ii) specify/update their QoT requirements, (iii) schedule computation, sensing and actuation by/at a reference time instant, (iv) timestamp events and (v) get latency estimates between a pair of nodes on a timeline. Note that for an application involving distributed coordination, latency estimates give a good idea of how far into the future actuation commands should be scheduled.

All API calls return the QoT actually delivered to the application, providing the ability to adapt to changes in the QoT. Thus, the Quartz API is designed to provide a core set of capabilities which are useful to applications relying on a shared notion of time to achieve coordination.

Listing 5.1 shows a simple application written using Quartz's Python API binding. The sample application binds to a timeline with an accuracy and resolution requirement of 1ms each. The application then periodically wakes up every second and reads the time. This is indicative of a collection of periodic time-triggered application components which each wake up at their own specific time instants to perform some coordinated action. Similarly, we can also envision event-driven applications which, in response to an event, capture a timestamp of the event. Such event timestamps can be captured using a callback function facilitated by the **`timeline_timestamp_events`** API call.

Listing 5.1: Simple Periodic App using the Quartz API

```
1  def main_func(timeline_uuid: str, app_name: str):
2    # Initialize the TimelineBinding class as an app
3    binding = TimelineBinding("app")
4    # Bind to the timeline with 1ms accuracy and resolution
5    ret = binding.timeline_bind(timeline_uuid, app_name, 1ms, 1ms)
6    if ret != ReturnTypes.QOT_RETURN_TYPE_OK:
7      print ('Unable to bind to timeline, terminating ....')
8      exit (1)
9    # Set the Scheduling Period and Offset (1s and 0ns repectively)
10   binding.timeline_set_schedparams(1000000000, 0)
11   while running:
12     # Wait until the next period
13     binding.timeline_waituntil_nextperiod()
14     # Do Something -> Read the time with the uncertainty
15     tl_time = binding.timeline_gettime()
16     print('Timeline time is       %f' % tl_time["time_estimate"])
17     print('Upper Uncertainty is   %f' % tl_time["interval_above"])
18     print('Lower Uncertainty is   %f' % tl_time["interval_below"])
19   # Unbind from the timeline
20   binding.timeline_unbind()
```

5.2.2 Quartz: Architecture & Implementation

To enable time-based geo-distributed applications at scale and deliver Time-as-a-Service, Quartz is tasked with the following primary objectives: (i) maintaining the notion of a timeline at geo-distributed scale, (ii) meeting application-specific QoT requirements with respect to the chosen timeline reference, and (iii) computing QoT estimates with respect to the chosen timeline reference. While meeting the above objectives, Quartz is also tasked with optimizing system resources by *merging* multiple timelines under the hood, based on application requirements and how they are deployed.

Given the above objectives, Quartz specifically needs to overcome the following challenges (i) ***scalability***: both geographical and quantitative, (ii) ***autonomy***: the system should autonomously adapt to application demands and faults, (iii) ***portability***: easy to deploy and manage, and (iv) ***ease of development***. Challenges (i) and (ii) are heavily influenced by the architecture, while (iii) is a function of the implementation, and (iv) is a function of the API.

A hierarchical architecture is one approach to both scalability and autonomy. Therefore, Quartz features a 3-tier hierarchical architecture with services which operate at the following tiers:

1) A **Node** represents any single computing node/device (virtual or physical) with an independent clock.

2) A **Cluster** represents any administrator-defined set of networked nodes which can communicate with each other. An example cluster is a set of nodes connected over a LAN. Note that a node cannot belong to more than one cluster, since each node has a single *independent* clock.

3) The **Global** scope represents the global set of clusters.

Based on the scope at which a timeline is discoverable by other nodes, we define two types of timelines:

1) A **Local Timeline** is discoverable only on nodes inside the cluster in which the

timeline is created. It is useful for applications with coordinating components restricted to the cluster scope.

2) A **Global Timeline** can be discovered by any node in the global set of clusters. A global timeline is useful for applications which have coordinating components spanning multiple clusters.

When a timeline is created, its type must be specified. This allows Quartz to choose an appropriate clock-synchronization protocol and virtual timeline reference, based on the application scope.

We implement Quartz using user-space micro-services, which are designed to run natively or as Docker [27] containers. Each service exposes an interface for exchanging information and receiving requests. Figure 5.2 illustrates the Quartz Architecture, and highlights the interactions between the various components through their exposed interfaces. We first describe each service's high-level implementation before stating how they provide different functions:

1) The **Timeline Service** is the interface through which applications interact with Quartz, i.e., most API requests are handled by the timeline service. It exposes a unix-domain socket (UDS)-based interface through which applications on the node can send requests to the service. It is also tasked with performing the bookkeeping of the timelines that exist on a node, the applications bound to each timeline, and the QoT requirements of each application and timeline. Therefore, the timeline service maintains timelines at the scope of a node, and hence, each node has its own timeline service.

2) The **QoT Clock-Synchronization Service** synchronizes the per-timeline clocks and computes the QoT estimates. Since every node has a hardware clock, which serves as a basis for per-timeline virtual clocks, each node has its own clock-synchronization service. Like the timeline service, it also exposes a UDS-based interface through which the timeline service can send it requests. In its current implementation, the synchronization service supports NTP [1], PTP [2] and Huygens [3] clock-synchronization protocols.

3) The **Coordination Service** is a distributed service responsible for maintaining

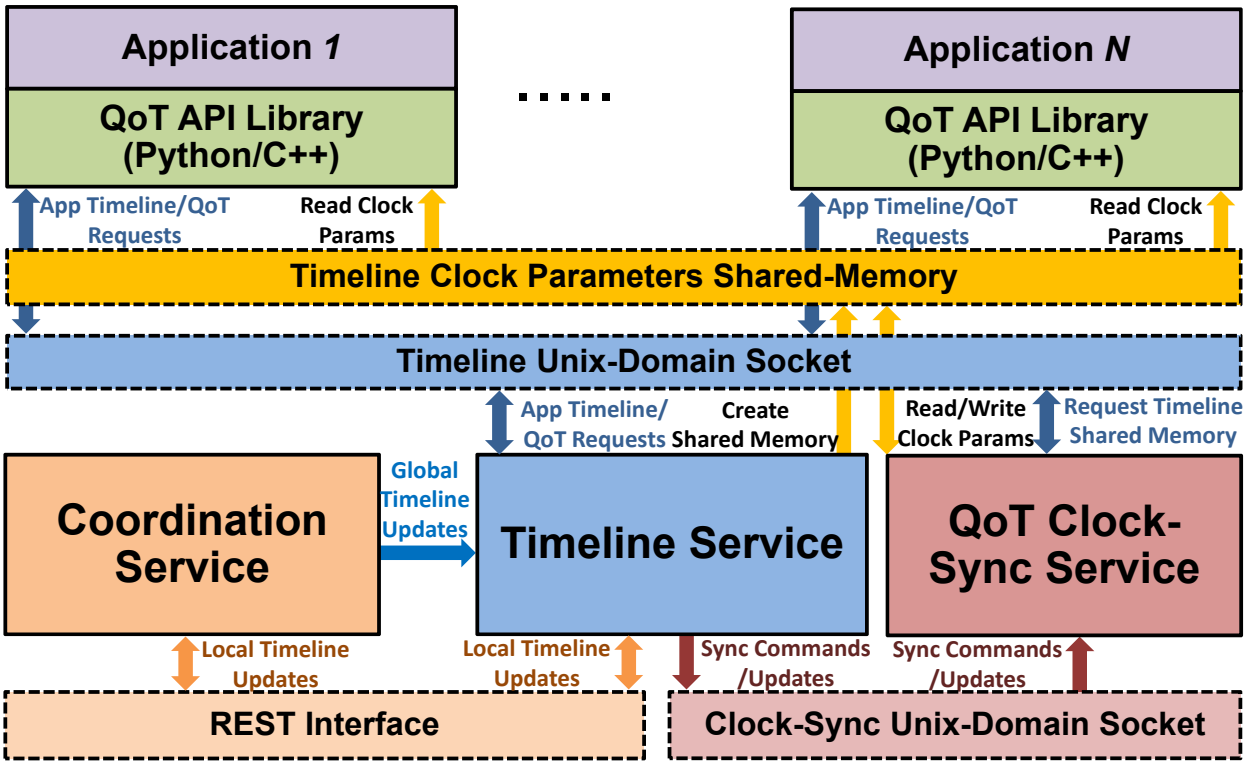


Figure 5.2: Quartz Time-as-a-Service. Solid boxes indicate components, and dashed boxes indicate interfaces.

timelines within the scope of a cluster. Hence, every cluster must have one active coordination service. Within a cluster, the coordination service helps each node discover other nodes on a timeline, and conveys QoT requirements across nodes. This information is used by each node's timeline service to orchestrate its node's clock-synchronization service, based on application QoT requirements. It exposes a REST API accessible to all the nodes within the cluster. The REST API allows the timeline service on each node to register (POST) timelines and its QoT requirement with the coordination service. This also allows timeline services on other nodes in the cluster to discover timelines (GET) and update (PUT) the most-stringent QoT requirement on a timeline.

4) The **Global Discovery Service** serves as Quartz's global book-keeper, and is tasked with maintaining timelines at the global scope, by allowing a cluster to *discover* the presence of other timelines and clusters bound to it. The discovery service maintains a key-value store of timelines and their relevant metadata along with the clusters associ-

ated with each timeline. It also provides an interface for cluster-specific coordination services to discover each other, and exchange timeline and QoT information. It is implemented using Apache Zookeeper [144], which provides a consistent and highly-available filesystem-like abstraction. The discovery service maintains a `/timelines` Zookeeper node, under which different timelines are registered. This allows cluster-specific coordination services to register the presence of timelines associated with their cluster, as `/timelines/<timeline-name>`. Under this timeline-specific Zookeeper node, a child node exists for each cluster participating in the timeline. In particular, the ability to (1) set *watches* on Zookeeper nodes: receive asynchronous notification on changes to a node or its children, and (2) *ephemeral* nodes: elements which disappear on a network disconnect, allows the coordination service to detect if another cluster has joined or left a timeline. Thus, Zookeeper is well-suited for the role of the global discovery service.

As may be expected, using a hierarchical architecture provides a very clear distribution of responsibilities. Therefore, even if higher-layer services (global or cluster-level) are temporarily lost, lower-layer services (cluster or node-level) can still continue to operate and provide essential functionality to applications.

Quartz Clocks: Quartz also features timeline-specific *clocks*, which are required for providing applications with their own shared notion of time. At the node scope, Quartz utilizes a core clock C_{core} [4] derived from a hardware clock, which maintains a monotonic free-running notion of time with undisciplined drift and offset. Each timeline-reference clock is maintained as a mapping from the core clock using the parameters tl_{drift} (drift correction), $core_{last}$ (the core-clock timestamp at the last synchronization event) and tl_{last} (timeline-reference timestamp at the last synchronization event). Using the current core timestamp, $core_{now}$, the timeline-reference time, tl_{now}, can be projected as follows:

$$tl_{now} = tl_{last} + tl_{drift} * (core_{now} - core_{last}) \tag{5.1}$$

A key proposition of Quartz is the ability to provide high-probability QoT bounds to

applications. Therefore, every timestamp provided to applications has its QoT bounds appended to it. At any instant of time, the timing uncertainty ϵ is given by the following equation:

$$\epsilon = tl_{bound} + tl_{skew} * (core_{now} - core_{last}) \tag{5.2}$$

where, tl_{skew} is a high-probability upper bound on the drift of the timeline-specific clock, and tl_{bound} is a high-probability upper bound on the offset of the timeline-specific clock. Note that the probability of these bounds should be configurable by a system designer. Therefore, given a QoT accuracy requirement Q, the probability of the bounds being invalid can be given by $P(\epsilon > Q)$. Therefore, for each timeline clock, with high probability $1 - P(\epsilon > Q)$, we can say that a timestamp $tl_{now} \in [tl_{now} - \epsilon, tl_{now} + \epsilon]$.

Hardware Timestamping: Most modern network interfaces have their own clocks and also provide the ability to timestamp some or all network packets in hardware at the physical layer [2]. This enables both accurate packet timestamping and clock synchronization, and is referred to as hardware timestamping. Therefore, Quartz also supports network-interface clocks C_{net}, and maintains an accurate mapping between the core clock and network clock(s).

5.2.3 Quartz: Inner Workings

Figure 5.3 provides a global view of Quartz, which highlights its hierarchical architecture. We now describe the inner workings of the services and their interactions.

Facilitating Low-Latency Clock Reads

From an application perspective, it is desirable that the timeline reference be read with low latency. To read a timestamp with its corresponding QoT, an application requires the current core-clock timestamp along with the timeline-projection and QoT parameters (Equations 5.1 & 5.2). Therefore, for each timeline, the timeline service creates a shared-memory region which holds the timeline projection and QoT calculation parameters.

Applications can request to map this shared-memory region, with read-only privilege, into their own virtual-memory space. Thus, by reading the core clock and applying the timeline projection parameters from shared memory, an application can read the timeline reference with low latency. In Quartz, we choose the Linux real-time clock (`CLOCK_REALTIME`) as our core clock, as it is available on all Linux systems, and can be read with low latency from user space [145]. Note that applications obtain read-only access to the timeline-clock shared memory, which prevents malicious applications from modifying the parameters held in shared-memory.

Handling Application Requests

Quartz provides a library implementation of its API which helps applications make requests, and removes the complexity of directly interacting with the timeline service. The API calls are stylized as remote procedure calls (RPCs) made by the application, and executed on the timeline service. However, only API calls related to (i) binding/unbinding from a timeline, (ii) updating timeline QoT requirements, and (iii) getting latency estimates between a pair of nodes, need to be handled by the timeline service. All other API calls related to scheduling sensing/computation/actuation, and time-stamping events are handled internally by the library in the context of the application process. Our initial version of Quartz implements C++ and Python bindings. However, the API can be generalized to any programming language which supports socket programming and shared memory. We now describe how Quartz handles application requests.

Timeline Creation/Deletion: When an application binds to a timeline, the information is sent to the timeline service using the API. If the timeline does not exist on the node, the timeline service creates an instance of the timeline. This instance keeps track of all applications on the node bound to that timeline, and the instance is deleted when no active bindings exist. The timeline service also checks if the timeline exists at the coordination service (GET), and if not, it registers the timeline at the cluster scope (POST). If the timeline exists at cluster scope, then the timeline service updates (PUT) the QoT

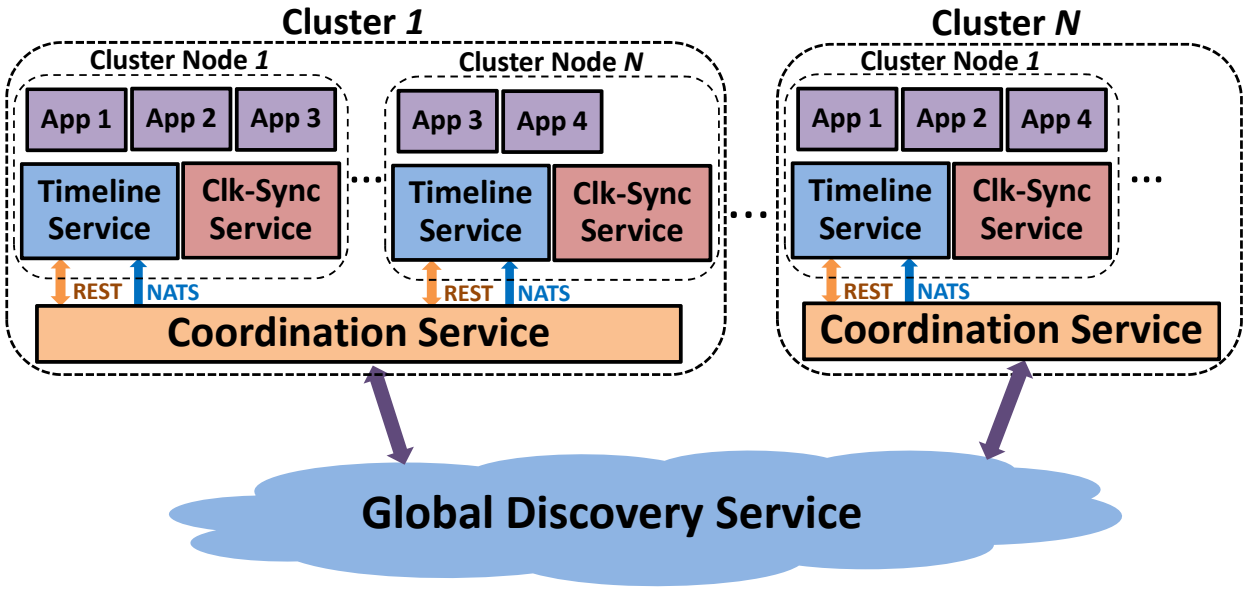


Figure 5.3: Quartz Time-as-a-Service at global scope

requirements, if they are more stringent than the timeline's most stringent existing QoT requirements. Similarly, the coordination service updates (creates) the timeline on the global discovery service at global scope if it exists (does not exist). A similar chain of events occurs for timeline deletion. The timeline service also creates a per-timeline shared-memory clock used to hold the timeline projection and QoT-estimation parameters. This shared-memory region is passed to the clock-synchronization service, which updates the projection and QoT-estimation parameters to synchronize the local timeline clock to the timeline reference.

Event Timestamping: Since Quartz is designed for containerized applications, there are three types of possible events: (1) software events timestamped by the system clock, (2) network events timestamped by the system clock (software/kernel timestamping) or the network-interface clock (hardware timestamping), and (3) externally-timestamped events on a sensor. While events of type (1) and (2) are commonly observed in software systems, events of type (3) are most likely to be observed in embedded systems.

To support time-stamping software and network events, the Quartz clock-synchronization service maintains a mapping between the core and network clocks

(if hardware timestamping is supported), along with the projection from the system (core) time to the timeline reference. Whenever these projection parameters are updated, the clock-synchronization service publishes them using NATS [146], which provides a publish-subscribe-based communication mechanism. The Quartz API library subscribes to these projection parameters and maintains a ring buffer of the last n projection parameters. Based on the incoming event system/network timestamp, the Quartz API library chooses the appropriate parameters from the ring buffer to project the event timestamp to the timeline reference. This new projected timestamp also contains a QoT estimate. To utilize hardware timestamping, the clock-synchronization service container, and the container getting network-timestamped packets, must be run in superuser mode.

In some embedded systems with general-purpose I/O (GPIO) pins, some pins have the ability to detect a voltage-change event and record (or *capture*) a corresponding hardware-timer value. This voltage-change event can also be triggered by a sensor. Through appropriate transformations, this hardware-timer value can be mapped to a timestamp on a timeline. In Quartz, we expose all such timestamping hardware using the Linux `ptp_clock` [147] abstraction. These clocks expose an I/O control (`ioctl`) interface over a `/dev/ptpX` character device, where X is a non-negative integer. To access this character device from the context of a Docker container, it needs to be mapped into the container at startup [148]. Additionally, most devices also require superuser privileges to access them. Therefore, there should be some higher-level admission-control service which decides if a container can access a device, and which then maps the device into the container's file-system at startup. Note that Quartz does not provide this functionality. However, on application startup, the Quartz API library enumerates all the `/dev/ptpX` devices available in the container's file-system, and exposes them to the application using its `timeline_timestamp_events` API call. In the background, Quartz uses the Linux `ptp_clock` headers and API to interface with the `/dev/ptpX` character device.

Event Scheduling: Scheduling an application on a *timeline* is important for execut-

ing distributed tasks/actuation synchronously. Therefore, the Quartz API library provides the ability to schedule events after a fixed time instant or duration on the timeline reference, in the form of *wait-until* calls, which suspend the application until a specified time instant. The library implements event scheduling internally, and schedules all events on the core clock. Therefore, the timeline-projection parameters are used to translate a scheduling request on the timeline reference to the core clock. Given that Quartz uses the Linux real-time clock (`CLOCK_REALTIME`), the Quartz API Library internally uses the existing `clock_nanosleep` POSIX API to schedule computation/actuation on `CLOCK_REALTIME`.

Latency Estimates: Consider a controller node sending out timestamped actuation commands to an actuator. From the controller's perspective, knowing a high-confidence end-to-end latency estimate between the planner and the endpoint gives it a good idea of how far into the future actuation commands should be scheduled. An end-to-end latency estimate characterizes the latency incurred in sending a message from user space to another application. To request a latency estimate, an application must first request (**`timeline_reqlatency`**) for the latency to a specific node on the timeline to be computed. This request is issued to the timeline service, which translates a node's unique name on a timeline to its corresponding IP address (using the coordination service). The Quartz API library then creates a new thread in the application context which uses *ICMP* packets (similar to the `ping` utility [149]) to compute the end-to-end latency. This latency measurement is projected to the timeline reference clock. The application also specifies the number of measurements n used to calculate a latency estimate, as well as the percentile value p which should be returned. Subsequently, the application can read the estimated latency using the **`timeline_getlatency`** call. Quartz uses a sliding window of the last n measurements to return the p^{th} percentile latency estimate.

Clock Synchronization and QoT Estimation

Quartz features a flexible implementation allowing integration with multiple clock-synchronization protocols over IP-compliant networks. Our implementation utilizes NTP [1], PTP [2] and Huygens [3] clock-synchronization protocols, and avoids re-inventing the wheel. This is because existing protocols like NTP and PTP are well-tuned for modern hardware, and are based on standards and implementations that have evolved and been refined over time. Meanwhile, Huygens is a recently-proposed protocol for data centers [3].

However, unlike traditional clock-synchronization protocols which are best-effort, Quartz monitors the delivered QoT to check if it is within the application-specified limits, and orchestrates the synchronization-protocol parameters to meet them. We now describe how Quartz provides an *autonomous* clock-synchronization service, which dynamically responds to application QoT requirements as well as external changes (network disconnections, changes and load). Quartz is autonomous in the sense that, based on application requirements, it chooses: (i) an appropriate protocol, (ii) an appropriate clock reference, and (iii) appropriate clock-synchronization tuning parameters. We first describe how Quartz synchronizes clocks for global timelines, and subsequently local timelines.

Global-Timeline Clock Synchronization: As global timelines can potentially span multiple clusters in different geo-distributed regions, the simplest way to maintain a shared notion of time is to synchronize all clocks to a common reference. To do so, Quartz uses the Network Time Protocol (NTP) [1] to synchronize the global timeline reference to Universal Coordinated Time (UTC). We use the *chrony* [150] implementation of NTP, which synchronizes the local clock by communicating with a set of NTP servers, and choosing the best source as the reference clock [150]. However, traditional NTP clients are often either configured using default or application/topology-specific tailor-made configurations. As Quartz is autonomous and aware of application requirements,

it responds to application demands by dynamically configuring NTP.

At startup, the Quartz timeline service automatically creates a default global timeline, and starts an NTP synchronization session on the synchronization service. Since, in our implementation, all global timelines follow a single clock reference, it suffices to maintain a single set of timeline-projection parameters for all global timelines. When an application binds to a global timeline specifying its timing requirements, Quartz checks if the application QoT requirement is being met. As all global timelines follow UTC, the QoT requirements of a global timeline are always defined with respect to UTC. Therefore, we use the *root dispersion* and *clock skew* values provided by NTP, which give a conservative estimate of how *far* or uncertain the clock is relative to UTC, to obtain a node's QoT. If the application QoT requirements are not being met, then Quartz tries to (i) either modify the synchronization rate, or (ii) if the root dispersion corresponding to the chosen reference indicates that the QoT requirements cannot be satisfied, it picks a new server from the pool of NTP servers. For a newly-created timeline, if the chosen server is able to deliver the desired QoT, Quartz registers this server with the cluster-specific coordination service, which in turn registers it with the discovery service. This chain of events allows other nodes on the same timeline, at both cluster and global scope, to select the same server as one of their reference sources. Thus, we ensure that nodes on the same timeline have a similar set of clock references. If available, Quartz also automatically chooses network-interface hardware time-stamping. Figure 5.4 presents a flow chart illustrating how we make NTP adaptive.

Local-Timeline Clock Synchronization: As local timelines are constrained to the cluster scope, we utilize PTP or Huygens (based on the system configuration) to synchronize the clocks in the cluster. When a local timeline is created on a node, the timeline service requests the synchronization service to start a timeline-specific synchronization session, and monitors the delivered QoT.

Precision Time Protocol (PTP) [2]: If PTP is the configured local-timeline protocol, then there is no need to choose a reference, as PTP automatically chooses a reference using

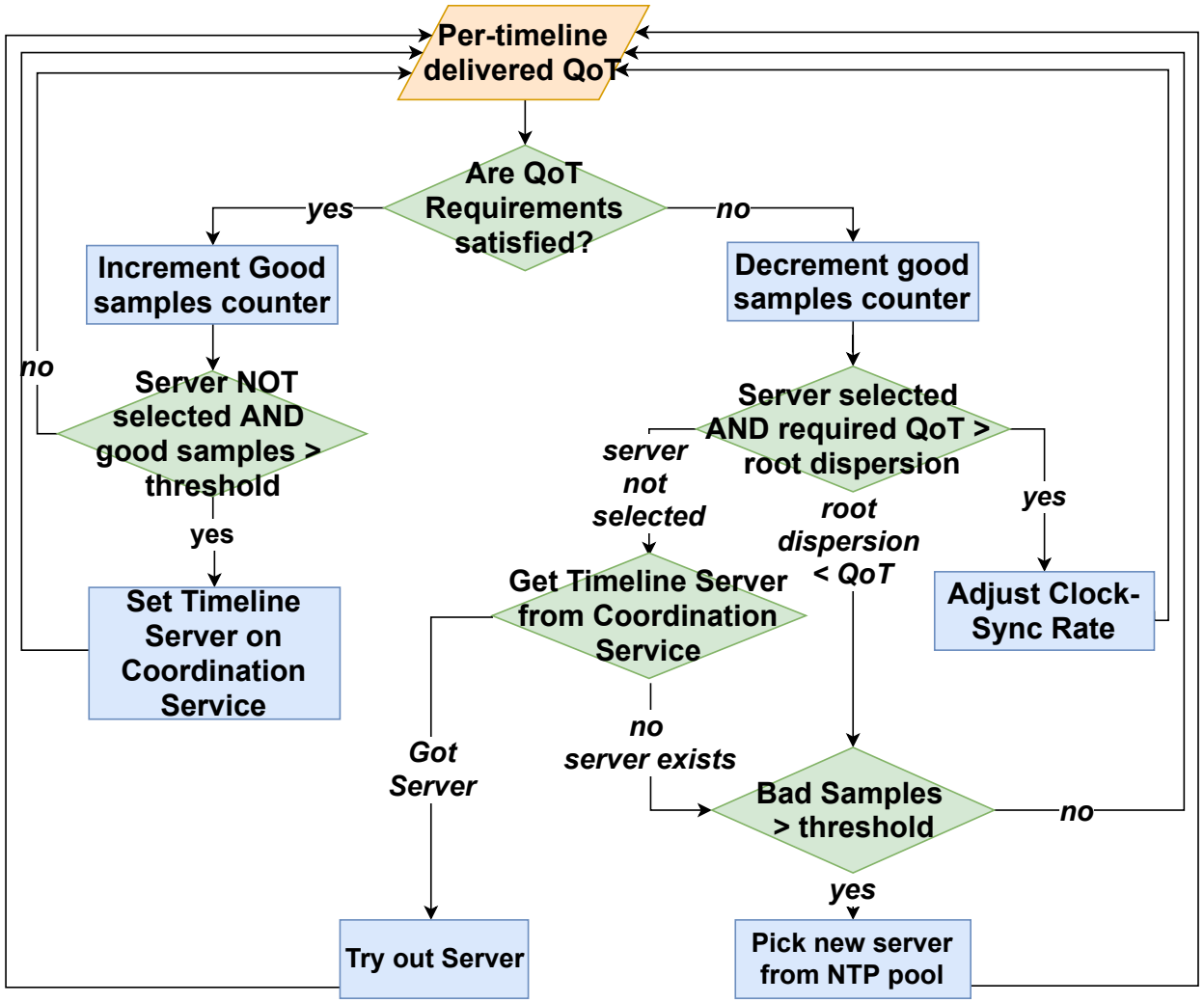


Figure 5.4: Adaptive NTP: Server Selection & Rate Adaptation

the best-master clock-selection (BMC) protocol [2]. However, Quartz modulates the PTP synchronization rate (message-exchange frequency) in order to match the application QoT requirements to the delivered QoT. Quartz uses the *linuxptp* [141] implementation of the PTP standard.

Huygens [3]: is the state-of-the-art protocol well-suited to operate at cluster scale. Huygens uses a mesh of probes, which estimates the offsets between pairs of nodes using a Support Vector Machine (SVM). Based on the probe-mesh topology, the pair-wise offsets are sent to a centralized server, which uses the network effect to calculate the final node offsets with respect to a pre-defined in-cluster clock reference. As Huygens is not open-source, we have written our own implementation which consists of three major

components: (1) per-node probe client-server pair which compute the pair-wise offsets and periodically publish the offsets using NATS, (2) per-cluster offset-calculator which calculates and publishes the final offsets by subscribing to the pair-wise offsets, and (3) per-node offset-receiver which subscribes to the final offsets.

If Huygens is the configured protocol, then the clock synchronization (1) topology, (2) rate and, (3) clock reference can be configured. Whenever a local timeline is created on a node, a *unique* offset-receiver is started per-timeline (allowing per-timeline clock-references), while the probe mesh and the offset-calculator are started at the first time a local timeline is created on the cluster. To meet the application-specified QoT, Quartz modulates the probe-mesh frequency, while monitoring the delivered QoT. In this initial version of Quartz, given that Huygens is designed to operate with a centralized server, we statically define the clock-synchronization topology and the master reference. However, future extensions can make the selection of both topology and master dynamic.

QoT Estimation: To estimate the QoT for local timelines using PTP or Huygens, the timing uncertainty relative to the local-timeline-reference needs to be computed by each node. In our implementation, we utilize the methodology proposed in **[?]** to compute the timing uncertainty. The proposed approach takes in a sliding window of n samples of the clock frequency-drift and offset (computed by the clock-synchronization protocol). After estimating the distribution of their variances, it computes a high-probability upper-bound on the clock offset and the drift, which can be used to estimate the QoT (Equation 5.2). Both the number of samples n and the confidence probability of the bounds can be configured.

Adaptive Synchronization Rate: Unlike NTP, both PTP and Huygens are master-driven synchronization protocols. This means that the master node drives the synchronization rate. For example, in PTP, the master clock reference sends periodic multi-cast `SYNC` packets to all the slaves at a pre-determined rate. The slave nodes then respond with follow-up packets, and hence, the master controls the rate of clock synchronization. Note that, having a single rate for the entire network implies the node with the tightest QoT

requirements holds significant influence on the synchronization rate. On the other hand, NTP is a client-driven protocol, and each client can independently decide and adapt its clock-synchronization rate by initiating a synchronization-request with an NTP server(s). Therefore, for both PTP and Huygens, we employ a similar clock-synchronization rate-adaptation strategy. Each node in a timeline periodically publishes its current delivered QoT on a particular timeline-specific topic using the NATS publish-subscribe mechanism. The master node listens to all the slave nodes, and tries to configure the synchronization rate to try to meet the QoT requirements of all the nodes on the timeline. Each master node has a protocol-specific lower and upper bounds on the rate of packets it can send. At the start, the master sends a burst of packets to quickly synchronize all the clocks. Subsequently, the master reduces its rate to the recommended protocol-specific rate. Based on whether the QoT requirements are being met or not, the master can gradually increase or decrease its synchronization rate.

Our entire implementation is open source and the source code along with the instructions to build and deploy can be found at: https://bitbucket.org/sandeepdsouza93/quartz/. We now illustrate how *TimeCop* (Section 5.1) is deployed using Quartz.

5.2.4 Enabling TimeCop with Quartz

To demonstrate TimeCop, since we do not have ready access to real traffic controllers in a city, we simulate a city-scale traffic scenario with multiple intersections, using the open-source SUMO traffic simulator [151]. We use TraCI [151] to interface with the simulation, and ensure that each time-step in the simulation mirrors the flow of time in the real world. Using TraCI, we expose each intersection as MQTT [152] endpoints which (i) periodically publish intersection sensor state – the number of vehicles queued per-incoming lane in the last period, and (ii) listen for commands – the next phase of the traffic signals at the intersection. Note that using MQTT decouples the simulation logic from the controllers.

Each containerized intersection controller is deployed using the Nutanix Xi IoT [153] platform which makes it easy to seamlessly develop, deploy, monitor and manage distributed IoT applications across multiple edge devices. Each controller gets the intersection state by subscribing to the MQTT endpoints corresponding to the intersection. The controller is based on deep reinforcement-learning [154], which uses the current intersection state to dynamically decide the next phase of the traffic signals at the intersection. The controller also periodically receives timestamped state from adjacent intersections, which it uses to improve traffic flow in coordination with other intersections. The chosen phase is published to the intersection MQTT endpoint listening for commands. Each intersection controller uses Quartz to bind to the *traffic-management* timeline with a QoT requirement of +/-1 ms, while Quartz ensures that all controllers bound to the timeline share the same notion of time with the desired QoT specification. Therefore, Quartz can be useful for building large-scale distributed-coordination applications.

The source code to build and deploy *TimeCop* can be found at: `https://bitbucket.org/sandeepdsouza93/traffic_app/`

5.3 Evaluation

We now evaluate the performance and scalability of Quartz. We first assess the accuracy delivered by the clock-synchronization protocols that Quartz supports: NTP [1], PTP [2] and Huygens [3]. For these protocols, we consider different time-stamping options (hardware/software) and platforms. In particular, we consider two embedded/edge-form-factor platforms: Intel NUC [155] and Beaglebone Black (BBB) [113]. Secondly, we highlight the ability of Quartz to adapt to application-specific QoT requirements, and accurately estimate the delivered QoT. Lastly, we evaluate the scalability of Quartz by creating a prototype geo-distributed-scale deployment.

5.3.1 Measurement Testbed

To perform clock-synchronization-related micro-benchmarks we have setup a testbed consisting of multiple Intel NUC (dual-core Intel Core i3, 8 GB RAM) and Beaglebone Black (uni-core ARM Cortex-A8, 1 GB RAM) nodes. The two platforms are representative of both embedded (BBB) and edge-computing (NUC) devices. As time-stamping network events is key to clock-synchronization, the NUC supports hardware timestamping of all UDP packets and the BBB supports hardware timestamping of PTP-compliant packets, as well as software timestamping of all packets. While both platforms are capable of running Docker containers, only the Intel NUC supports the Kubernetes container-orchestration engine [156].

The test-bed consists of two LANs: *greenwich* and *roseline*. Greenwich has two clusters (i) *NUC-Amethyst*: Kubernetes cluster with 4 NUCs, and (ii) *BBB-Citrine*: 4 BBBs with Docker; and an event generator *BBB-Onyx*. The event generator creates events (UDP packets, or voltage-change on a hardware pin), which serve as opportunities for other nodes to timestamp. By comparing the timestamps of a common event, the offset between two clocks can be measured. Roseline has one cluster *BBB-Ametrine*: 4 BBBs with Docker.

We now describe each of the cluster types and their utility.

1) The **BBB Clusters** are used to benchmark the performance of (i) Huygens and PTP with hardware timestamping, and (ii) NTP with software timestamping. The BBB hardware strictly restricts hardware timestamping to PTP-compliant multi-cast packets sent/received on port 319 over 4 prescribed multi-cast IPs [157]. This constrains us to performing Huygens micro-benchmarks with not more than 4 nodes. However, the BBB have GPIO pins which allow 42ns-resolution timestamping on a rising or falling edge (generated by the event-generator BBB-Onyx). This allows us to externally measure the offset between two clocks and validate our implementation. In addition, having two BBB clusters on different LANs (citrine on greenwich, and ametrine on roseline) also enables

Table 5.1: NTP [1] Accuracy Micro-benchmarks (μseconds)

Platform	Timestamps	Cluster	Stratum	Max	Mean	Std. Dev
NUC	HW	Intra	1	4267	**380**	**633**
	HW	Intra	2	12607	2480	3351
BBB	SW	Intra	1	1638	**542**	**245**
	SW	Intra	2	5855	2380	717
	SW	Inter	1	2127	**929**	**553**
	SW	Inter	2	6033	3582	1032

clock-synchronization accuracy measurements between the two LANs.

2) The **NUC Cluster** is used to benchmark the performance of NTP, PTP and Huygens with hardware timestamping. The NUC features a desktop-class processor and a low-cost gigabit network interface [158] which supports hardware timestamping. As the NUC does not have external pins, the synchronization accuracy cannot be externally measured. Instead, we use the event generator (BBB-Onyx) to periodically generate multi-cast UDP packets, which the NUC timestamps in the network-interface hardware. We use these timestamps (after applying the timeline-projection parameters) to compute a safe upper bound of the offset between the two clocks. To ensure that the multi-cast event reaches all the NUCs as *simultaneously* as possible, the event generator is connected to the same switch as the NUC, and short cables of the same length are used to connect each NUC to the switch.

5.3.2 Quartz: Clock-Synchronization Accuracy

We now present micro-benchmarks to evaluate the performance of NTP, PTP and Huygens in various scenarios based on (i) timestamping capability (hardware/software), (ii) platform (NUC/BBB), and (iii) server stratum (for NTP). Our micro-benchmarks are intended to provide a glimpse of the *best-effort* accuracy deliverable by a protocol on a given platform. This helps us to gain insights required to autonomously select an appropriate protocol and configuration within Quartz. The accuracy (modulus of the measured offset) obtained from our test-bed is specified in micro-seconds (μs).

Table 5.2: PTP [2] Accuracy Micro-benchmarks (μseconds)

Platform	**Timestamps**	**Rate** (s)	**Max**	**Mean**	**Std. Dev**
NUC	HW	1	183	31	113
	HW	2	220	24	32
	HW	4	13	**9**	**2**
BBB	HW	1	14	**2**	**3**
	HW	2	39	8	7
	HW	4	39	5	7

1) **NTP**: The NTP micro-benchmark accuracy results are summarized in Table 5.1. In all the experiments, we utilize publicly-available NTP pool servers, and each node can pick its own server. In Section 5.2.3, we define all global timelines relative to UTC. Therefore, not constraining nodes to pick a single server avoids the server from becoming a single point of failure or a performance bottleneck, which is useful for maintaining timelines at global scale. This provides us with a good estimate of the accuracy achievable in real-world deployments without the need for custom NTP infrastructure. However, better accuracy can be achieved using custom NTP-server deployments. If we compare the measured accuracy based on platform type or network-timestamping capabilities, no significant differences are observed. This is because (i) NTP requires few resources and can synchronize clocks efficiently even on low-power platforms like the BBB, and (ii) most NTP servers do not support hardware timestamping on their end. On the other hand, we observe that, regardless of the platform, the choice of server (stratum) plays an important role in the accuracy obtained. This is because lower-stratum NTP servers track UTC with lower error. Thus, even choosing different stratum 1 servers can yield sub-millisecond accuracies across different LANs (Inter). Thus, NTP is well suited for global-scale applications which have QoT requirements in the order of 100s of μs to several ms.

2) **PTP**: Both platforms support hardware timestamping of IEEE 1588 PTP packets, and the accuracy results are summarized in Table 5.2. The network we utilize is not PTP-compliant and does not correct for queuing delays, which is mostly true for real-world

networks. For both platforms, we observe that PTP at LAN-scale can yield accuracies in the order of 1-100 μs. This is primarily due to the use of hardware timestamping, and the fact that the Linux kernel natively-supports PTP. On the NUC, decreasing the synchronization rate causes a slight increase in accuracy. In contrast, on the BBB, a lower synchronization rate yields marginally better accuracy. Thus, a faster rate does not always imply better accuracy. The Allan intercept of the clock [159], an indicator of clock stability, influences the optimal rate. Therefore, choosing the correct rate *autonomously* is useful in achieving application-specified levels of QoT.

3) **Huygens**: We benchmarked Huygens on both platforms, and the accuracy results are summarized in Table 5.3. For both platforms, we consider a toy deployment of 4 nodes (in their clusters) with the probe-mesh pairs setup to form a 4-node loop. We observe that Huygens at LAN-scale can yield accuracies in the order of 100s of μs. The values in the table are for a pair of nodes separated by one hop in the probe mesh, while the values within parentheses are for a pair of nodes separated by two hops. Huygens relies on exchanging 10-100s of packets between nodes every second, and is designed for data-centers and not low-cost hardware. In both of our platforms, while using hardware timestamping, we observed significant timestamping errors at the network interface. This was especially severe for the BBB, which incorrectly orders/loses timestamps when packets arrive rapidly. Hence, we observed accuracies of the order of a few seconds, as the BBB NIC is only designed to timestamp PTP packets arriving at a rate of about 1-4 packets per second [157]. We also run Huygens with software (kernel) timestamping, and observe that it yields an accuracy in the order of 100s of μs, and the synchronization session is stable. This obtained accuracy is in line with the resolution provided by kernel timestamps.

Therefore, while both NTP and PTP are well-suited to run on low-cost platforms, Huygens is better suited for resource-rich settings.

Table 5.3: Huygens [3] Accuracy Micro-benchmarks (μs)

Platform	Timestamps	Rate (ms)	Max	Mean	Std. Dev
NUC	HW	10	401 (1596)	294 (1099)	21 (501)
	HW	100	405 (382)	**104** (**105**)	**64** (**75**)
	SW	10	1835 (1205)	294 (252)	242 (163)
	SW	100	1251 (965)	234 (328)	259 (243)
BBB	HW	100	13000000	2000000	3000000
	SW	10	782	**170**	**153**
	SW	100	4593	1091	340

5.3.3 Quartz: Adaptiveness & QoT Estimates

The key proposition of Quartz is to provide Time-as-a-Service, and adapt to application-specific QoT demands. We now evaluate Quartz's ability to: (i) orchestrate clock-synchronization protocols to deliver application-specific QoT requirements, and (ii) report accurate QoT estimates to applications during transient (external disturbances) or permanent failure (network disconnect). We first focus on NTP, as it is our protocol of choice for providing TaaS at geo-distributed scale (global timelines). Subsequently, we benchmark PTP and Huygens for cluster-scale local timelines.

Global Timelines

Figures 5.5a, 5.5b, 5.5c and 5.5d showcase four scenarios, where two application components α_1 and α_2, on two different nodes (Node1 and Node2), each bind to a global timeline *gl_test*, specifying their QoT requirements of +/-1 ms relative to UTC. Each figure plots the measured offset between the two nodes, as well as the QoT estimate that Quartz provides to each application. As the QoT bounds presented for global timelines are always relative to UTC, the bounds can be lesser than the measured offset between two nodes. As mentioned in Section 5.2.3, all global timelines are maintained relative to UTC, and hence, we use only one NTP instance to synchronize all the global timelines. Quartz orchestrates and configures this NTP instance to meet application requirements.

Adaptivity: Figure 5.5a showcases Quartz's ability to adapt to application-specific QoT requirements. At time $t = 0$, α_1 on Node1 (NUC-Amethyst-1) binds to the time-

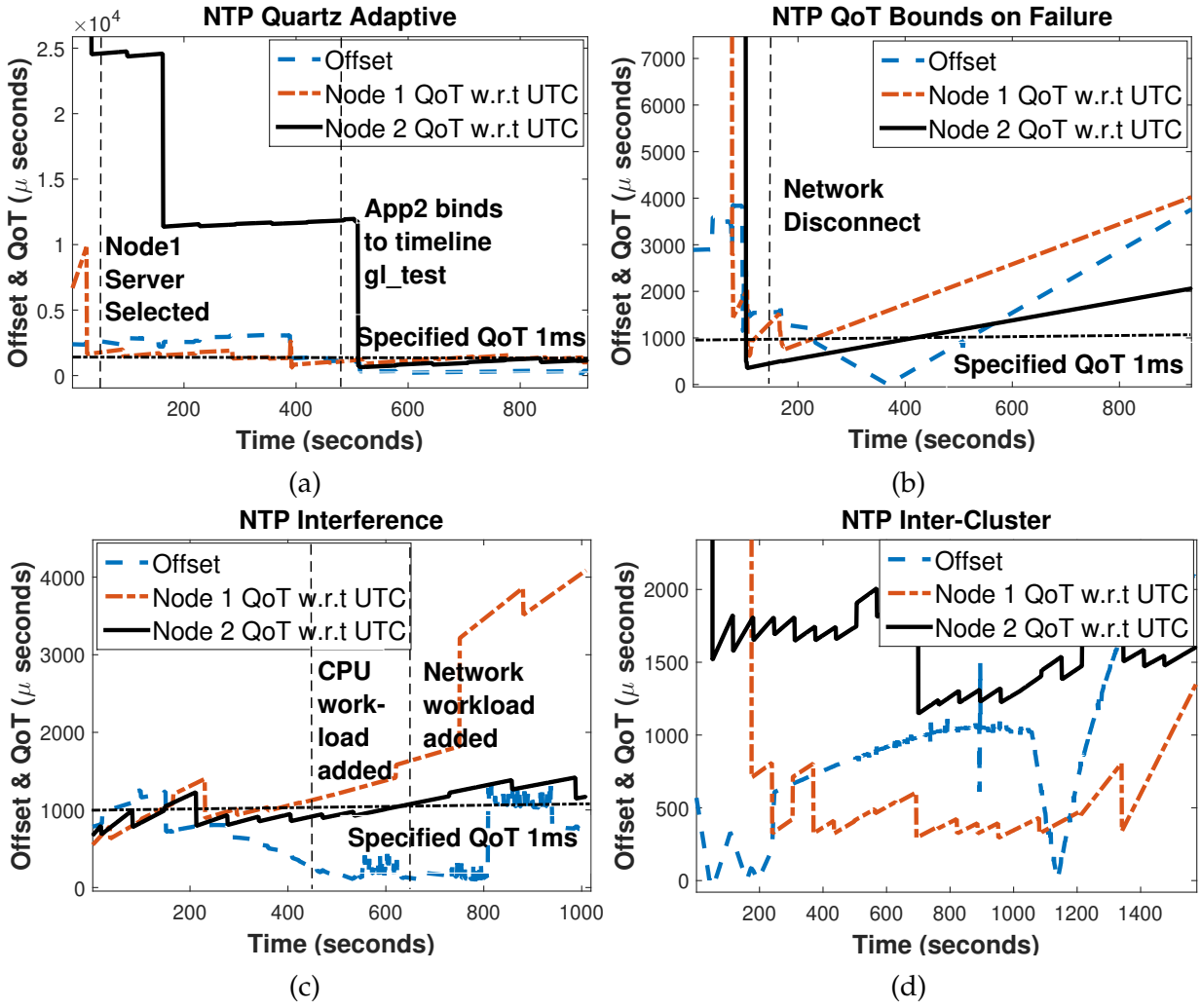


Figure 5.5: Quartz NTP: (a) Adaptive clock-synchronization, (b) QoT bounds on clock-synchronization failure, (c) Effect of CPU & network interference on QoT, and (d) Inter-cluster QoT estimation. Note the different y-axis on each plot

line *gl_test*. As the existing clock reference cannot satisfy α_1's requirements, Node1's synchronization service tries new servers from the NTP pool, until the first dashed line, when it selects a suitable server which meets α_1's requirements. At time $t = 500$, α_2 on $Node_2$ (NUC-Amethyst-2) binds to *gl_test*. As Node2's current reference cannot satisfy α_2's requirements, Node2's timeline service queries the cluster-scope coordination service for any known NTP servers being used by other apps on *gl_test*. As a server exists (registered by α_1), Node2's synchronization service selects it, and is able to meet α_2's QoT requirements. As a consequence, the offset between the two nodes reduces, and

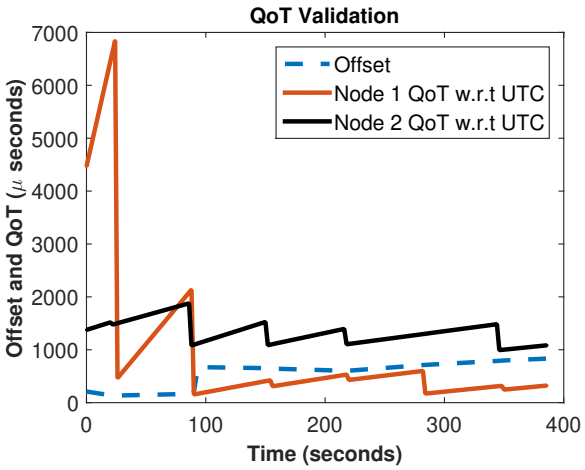


Figure 5.6: Validating the accuracy of the QoT bounds for NTP

this is reflected in the QoT bounds returned to α_2.

QoT-based Fault Detection: Figure 5.5b plots a network-disconnection scenario where clock synchronization is lost, as a node(s) is unable to communicate. At time $t = 180$, we simulate a network-disconnect/synchronization-service failure by killing the synchronization service on both Nodes1&2 (NUC-Amethyst-1&2). In this scenario, the API library (used by α_1 & α_2) uses the last-known QoT parameters to keep estimating the QoT (using Equation 5.2), until the clock is re-synchronized. As highlighted in Figure 5.5b, the bounds diverge linearly at the rate given by the upper bound of the clock drift (tl_{skew}). When the bounds exceed application-specified QoT requirements, the application is notified.

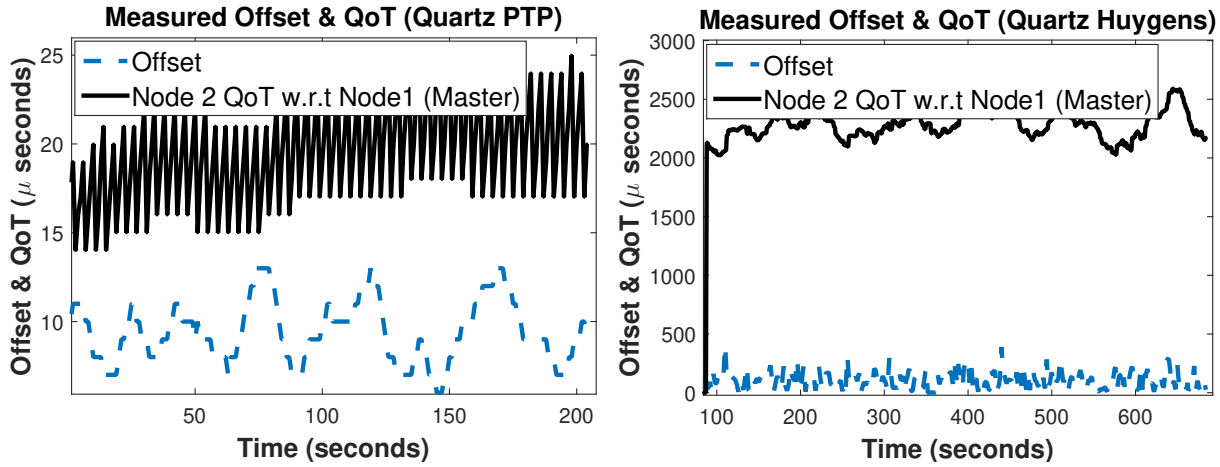
Resilience to CPU/Network Interference: Processing and networking resources are essential to clock synchronization. Figure 5.5c illustrates the effect of adversarial CPU and network-intensive workloads on the QoT and offset between two nodes (NUC-Amethyst-1&2). Between time $t = 500$ and $t = 700$, we introduce a CPU-intensive workload on Node1 using the *stress* tool [142]. The *stress* tool creates 10 CPU-intensive threads which nearly saturate the CPU on node1. Observe that, as NTP is a lightweight protocol, there is no significant effect on the measured clock-synchronization accuracy, and this is also reflected in the QoT bounds. At time $t = 700$, we introduce a network-

intensive workload on Node1 using the *iperf* tool [143]. The *iperf* tool fully saturates the network interface on Node1 with TCP traffic. Shortly after the network load is introduced, there is a degradation in clock-synchronization accuracy, as reflected by the ~4x increase in the measured offset between Nodes1&2. Note that the QoT bounds delivered to the application on Node1 also suddenly increase to reflect this degradation in clock-synchronization accuracy. Hence, Quartz detects transient changes in QoT due to anomalies or interference.

Inter-Cluster QoT Estimation: Computing accurate QoT estimates across clusters in different LANs is key to providing TaaS at geo-scale. Figure 5.5d plots the measured offset between two nodes (BBB-Citrine-1&BBB-Ametrine-1) in different LANs (*greenwich* and *roseline*), as well as the reported QoT. Additionally, to validate whether the QoT bounds are accurate, we also consider two nodes synchronized to different NTP servers. For this scenario, Figure 5.6 plots the measured offset between two nodes (BBB-Citrine-1&BBB-Ametrine-1) in different LANs (*greenwich* and *roseline*), as well as the reported QoT, while each of these nodes are synchronized to a different NTP server – `ntp-1.ece.cmu.edu and` and `ntp1.wiktel.com` respectively. For both Figures 5.5d and 5.6, as the QoT is defined relative to UTC, for the bounds to be valid, the sum of the two QoT bounds should not be less than the measured offset between the two nodes.

Local Timelines

Figures 5.7a & 5.7b plot the QoT estimates for local timelines, when using PTP and Huygens respectively. For both protocols, the QoT and offset of Node2 are defined relative to the timeline reference (Node1). Both sets of measurements were obtained using a pair of NUCs (NUC-Amethyst-1&2). For Huygens, we observed significantly higher QoT bounds than the measured offset. This is due to the high variance in the clock offset and drift measurements caused by hardware timestamping instabilities/errors in the network interface. Therefore, for local timelines, we focus on the PTP protocol. We utilize a pair of nodes in the *BBB-Citrine* cluster to perform experiments. As stated



(a) Quartz PTP (b) Quartz Huygens

Figure 5.7: Quartz QoT estimation: (a) PTP and (b) Huygens

before, the BBB have GPIO pins which allow 42ns-resolution timestamping of a voltage-change event (generated by the event-generator BBB-Onyx). We use this to externally measure the offset between two clocks.

Adaptivity: Figures 5.8a & 5.8b showcase Quartz's ability to orchestrate PTP to adapt to application QoT requirements. The left y-axis shows the measured offset and the estimated QoT, and the right y-axis shows the binary logarithm ($\log_2$) of the period of the PTP `SYNC` messages [2]. As described in section 5.2.3, Quartz modulates PTP's clock-synchronization rate to meet application QoT requirements. We consider α_1 on Node1 (BBB-Citrine-1) and α_2 on Node2 (BBB-Citrine-2) bound to the local timeline *test*. In both Figures 5.8 5.8a & 5.8b, α_1 on Node1 is elected as the timeline master-clock reference, and the application QoT requirements are set to (a) 10μs and (b) 5μs respectively. In Figure 5.8a, Quartz initially increases the rate to quickly meet the QoT requirements, and then slows down once the QoT requirements are met. Similar observations can be made for the case illustrated in Figure 5.8b. Note that for a multi-cast protocol like PTP, decreasing the synchronization rate can lead to significant reduction in network bandwidth consumed. Additionally, in case (b), sometimes during durations of high-synchronization rates, there can be timestamping instabilities, which cause the offset, and the delivered QoT to spike. We believe that this is due to an issue in the BBB

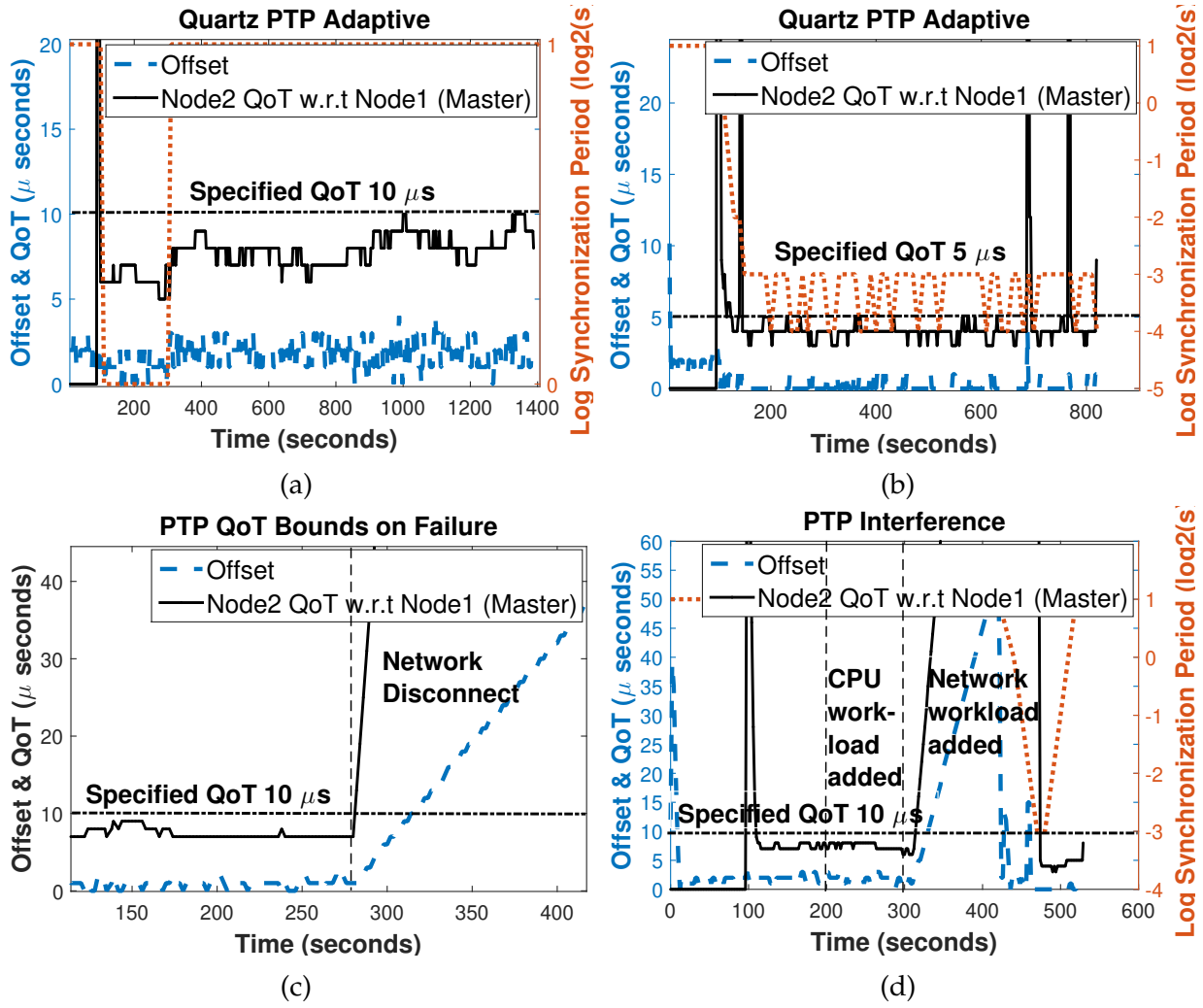


Figure 5.8: Quartz PTP: Adaptive clock-synchronization with QoT requirement (a) 10μs, (b) 5μs, (c) QoT bounds on clock-synchronization failure, and (d) Effect of CPU & network interference on QoT. Note the different y-axis on each plot

hardware-timestamping module.

QoT-based Fault Detection: Figure 5.8c plots a network disconnection scenario where clock synchronization is lost, as a node(s) is unable to communicate. At time $t = 280$, we simulate a network-disconnect failure by killing the synchronization service on Node2 (BBB-Citrine-2). Similar to the NTP case, the API library (used by α_2) uses the last-known QoT parameters to keep estimating the QoT (using Equation 5.2), until the clock is re-synchronized. As highlighted in Figure 5.8c, the bounds are diverged linearly at the rate given by the upper bound of the clock drift (tl_{skew}). When the bounds

Table 5.4: Continent-scale Scalability Results

Specified QoT (Accuracy)	**Worst Delivered QoT**	**Best Delivered QoT**
500μs	442μs	284μs
1ms	994μs	233μs

exceed application-specified QoT requirements, the application is notified and can enter a graceful-degradation mode.

Resilience to CPU/Network Interference: Figure 5.8d illustrates the effect of CPU and network-intensive workloads on Quartz PTP. At time $t = 200$, we introduce a CPU-intensive workload on Node2 using the *stress* tool [142] for 100 seconds, which fully saturates the CPU on the BBB. Since PTP is lightweight, there is no significant effect on the clock-synchronization accuracy, and the observed QoT bounds. At time $t = 400$, we introduce a network-intensive workload on Node2 using *iperf* [143] for 100 seconds. This saturates all the network bandwidth, and PTP packets cannot get through. Thus, the clock offset as well as the observed QoT diverges. Observe that, as soon as the network interference goes away, Quartz increases the PTP clock-synchronization rate to ensure that the delivered QoT quickly returns to the desired level (10μs).

Therefore, we conclude that Quartz adapts to application demands and external interference at both cluster and global scales.

5.3.4 Scalability

We now demonstrate the ability of Quartz to provide Time-as-a-Service at geo-distributed scale, by utilizing clusters deployed using Virtual Machines (VMs) hosted in the public cloud. Our experiments are meant to demonstrate scale, and hence we consider global timelines maintained using Quartz's Adaptive NTP clock-synchronization protocol. As we cannot externally measure the accuracy of clock-synchronization inside the VMs, we rely on the ability of our system to accurately provide QoT estimates, to check if different application-specified QoT levels can be achieved across all the geo-distributed clusters. We conduct two sets of experiments:

Table 5.5: Geo-distributed Scalability Results: Microsoft Azure

QoT Spec.	Region	Worst QoT	Best QoT	Average QoT	Fraction
500μs	east-us	506μs	200μs	327μs	0.98916
	central-us	504μs	216μs	354μs	0.98844
	west-europe	508μs	249μs	415μs	0.97398
	east-australia	NA	NA	NA	NA
	east-asia	NA	NA	NA	NA
1 ms	east-us	635μs	199μs	365μs	1
	central-us	568μs	140μs	293μs	1
	west-europe	640μs	307μs	476μs	1
	east-australia	1003μs	490μs	758μs	0.99076
	east-asia	1006μs	459μs	645μs	0.97398

1) Continental Scale: We deploy Quartz across 15 VMs running across three Amazon Web Services (AWS) [160] regions spanning the continental United States (5 VMs each in us-east-1 Virginia, us-east-2 Ohio and us-west-2 Oregon). Each VM is configured as a standalone Kubernetes cluster using the Nutanix Xi IoT [153] platform, which also helps deploy the Quartz micro-services as Kubernetes pods. In this experiment we deploy an application with 15 coordinating components, each deployed in one of the 15 VMs. Each application component binds to a common global timeline `gl_test`, and specifies its QoT requirement. This experiment gives us an idea of the accuracy that Quartz can achieve at continental scale, for a geo-distributed deployment on a single network backbone. We conduct this experiment over a period of 5 hours and consider two QoT-specification levels (required clock-synchronization accuracy): 500 μs and 1 ms. For a given specified QoT level, Table 5.4 summarizes the best and worst QoT level delivered by Quartz across the 15 geo-distributed clusters. As seen in Table 5.4, the best QoT represents the tightest accuracy bounds observed, and the worst QoT represents the loosest bounds observed.

2) Global Scale: We deploy Quartz across 20 Virtual Machines (VMs) spanning five continents and two public cloud providers. Our deployment consists of 10 VMs running in five Microsoft Azure (Azure) [161] regions (2 VMs each in east-us, central-us, europe-west, australia-east and asia-east), and 10 VMs running in five Google Cloud (GCP) [162] regions (2 VMs each in asia-east, asia-south, us-west, europe-north and

Table 5.6: Geo-distributed Scalability Results: Google Cloud

QoT Spec.	Region	Worst QoT	Best QoT	Average QoT	Fraction
500μs	asia-east	716μs	230μs	376μs	0.96025
	asia-south	886μs	214μs	390μs	0.94606
	us-west	501μs	184μs	289μs	0.99850
	europe-north	389μs	186μs	291μs	1
	south-america	1100μs	276μs	473μs	0.87861
1 ms	asia-east	648μs	292μs	426μs	1
	asia-south	813μs	237μs	484μs	1
	us-west	1009μs	224μs	542μs	0.99566
	europe-north	509μs	204μs	309μs	1
	south-america	746μs	277μs	458μs	1

south-america-east). Each VM is configured as a standalone Kubernetes cluster. In this experiment we deploy an application with 20 coordinating components, each deployed in one of the 20 VMs. Each application component binds to a common global timeline `gl_test`, and specifies its QoT requirement. This experiment gives us an idea of the accuracy that Quartz can achieve for a geo-distributed deployment. We conduct this experiment over a period of 5 hours and consider two QoT-specification levels (required clock-synchronization accuracy): 500 μs and 1 ms. Tables 5.5 and 5.6 summarize the best, worst and average observed QoT, along with the fraction of time the specified QoT requirements were satisfied, for the VMs deployed in Azure and GCP respectively.

For our continental-scale experiments on AWS (Table 5.4), we observe that Quartz can reliably deliver an accuracy level of 500μs. On the other hand, for our global-scale deployment across Azure and GCP (Tables 5.5 and 5.6), we observe that some nodes cannot achieve a QoT level of 500μs. This is especially true for the Azure nodes deployed in the east-australia and east-asia regions (values indicated by NA in Table 5.5). This is because Quartz is unable to choose an appropriate NTP server to satisfy the QoT specification of 500μs.

Note that the lowest-possible uncertainty with respect to Universal Coordinated Time (UTC), achievable by a client using a specific NTP server depends on the server's: (i) stratum [1], i.e., how *closely* it tracks UTC, and (ii) the round-trip network latency be-

tween the server and the client. For example, if Quartz chooses a low-stratum server located in the United States (US), which tracks UTC accurately, then the high round-trip latency between the server in the US, and a client in Australia will constitute the dominant factor in the clock-synchronization uncertainty or QoT reported by Quartz. This may prevent the QoT requirements of 500 μs from being met.

Thus, Quartz can maintain global timelines with sub-millisecond accuracy, while estimating QoT at geo-distributed scale.

5.4 Summary

Time is a key primitive for enabling coordination in distributed systems. In this chapter, we introduced Quartz which provides *Time-as-a-Service* to geo-distributed containerized applications, which coordinate using a shared notion of time. Based on the notion of Quality of Time (QoT) in conjunction with the timeline abstraction, Quartz exposes an API which simplifies the development of geo-distributed coordinated applications, and allows applications to specify their QoT requirements. Quartz orchestrates the underlying infrastructure to meet these application-specific requirements, and exposes the delivered QoT back to the application. Thus, time-based distributed-coordination applications can be fault-tolerant in the face of clock-synchronization failure.

Quartz features a modular architecture implemented using containerized microservices. This makes it scalable and easy to deploy on platforms ranging from embedded devices to the edge, and the cloud. Our evaluation indicates that Quartz adapts to application demands, and maintains a *timeline* across multiple geo-distributed nodes. We also demonstrated the utility of Quartz by using it in *TimeCop*, which uses a shared notion of time to coordinate traffic signals, and consequently vehicular-traffic flow at city scale.

Our realization of Quartz is open-source and supports Python and C++ applications, along with NTP, PTP and Huygens clock-synchronization protocols. While Quartz is most relevant for cyber-physical systems, the core concept of Time-as-a-Service is also

useful for distributed software applications, such as databases and logging systems. We strongly believe that the ability to request and observe application-specific QoT can be used to relax many of the stringent asynchronous assumptions associated with distributed systems.

Chapter 6

Sleep Scheduling for Energy-Savings in Multi-Core Processors

The emergence of edge computing and the Internet of Things (IoT) [12] have begun to place an increasing demand for computation on highly-connected and mobile devices. Platforms such as Google Glass and smartwatches are examples of IoE systems using multi-core processors. The need for high performance in an energy-constrained environment makes it necessary to investigate the use of various energy-management techniques for multi-core systems. To increase battery life, modern processors are equipped with a number of energy-management features. Primary among them are Dynamic Voltage and Frequency Scaling (DVFS) [41], and the use of low-power sleep states. The use of DVFS enables the processor to change its operating frequency and voltage, thereby reducing *dynamic switching power*. On the other hand, low-power sleep states use power gating and/or clock gating [43] to reduce *static leakage power* dissipation when the processor is idle. However, there is a minimum *round-trip* time associated with each of these low-power sleep states [42]. This round-trip time is longer for moving to and from lower-power states due to the overhead required for the main oscillator to startup and stabilize [42].

From a real-time systems scheduling perspective, it is critical that all tasks meet their

deadlines to ensure reliable system operation. The presence of such systems in resource-constrained and mobile environments also makes it essential that the system minimize energy consumption. As technology scales, the dominance of static power makes it necessary that scheduling techniques take advantage of built-in processor sleep states.

The focus of this chapter is on energy-efficient multi-core partitioned fixed-priority real-time scheduling techniques, which utilize the processor's deep-sleep state to reduce static leakage power, and hence save energy. The primary contributions described in this chapter are as follows:

- We propose an enhanced version of ES-RHS, named ES-RHS+, which has better schedulability properties than ES-RHS [42].
- We provide an energy-saving version of rate-monotonic (deadline-monotonic) scheduling, named *Energy-Saving Rate-Monotonic (Deadline-Monotonic) Scheduling* (ES-RMS and ES-DMS), which has better energy-saving guarantees than ES-RHS and ES-RHS+ for multi-core processors where all cores can only transition into deep-sleep state together.
- We present a new task-partitioning heuristic that increases synchronized sleep times, where all cores can only transition into deep-sleep state together.
- We prove that ES-RHS (ES-RHS+) is optimal for energy savings on multi-core processors where, each core can independently go into deep-sleep state, for any partition feasible under multi-core ES-RHS (ES-RHS+).

6.1 Energy-Saving Rate-Harmonized Scheduling

This section introduces the notation used in the context of uniprocessor ES-RHS. We then describe a version of ES-RHS with enhanced schedulability conditions, named ES-RHS+.

6.1.1 Notation and Background

Consider a taskset Γ consisting of n independent[1] periodic real-time tasks $\tau_1, \tau_2, ..., \tau_n$. Each task $\tau_i \in \Gamma$ can be characterized by $\{C_i, T_i, D_i\}$, where C_i is the worst-case execution time, T_i is the period, and D_i is the relative deadline from its arrival time. In this chapter, we assume that $D_i = T_i$, i.e., for each task, deadlines are implicit. The utilization of a task τ_i is given by $U_i = C_i/T_i$. Consider fixed-priority preemptive scheduling, with task priorities assigned using the rate-monotonic policy [29]. The taskset is listed in non-increasing order of task priorities such that $T_1 \leq T_2 \leq .. \leq T_n$. Each task has an initial arrival time of ϕ_i, such that its arrival times are $\phi_i, \phi_i + T_i, \phi_i + 2T_i, ...$ Without loss of generality, we assume that the initial arrival time of task $\tau_1, \phi_1 = 0$.

The family of *Rate-Harmonized Schedulers* [42] utilizes a periodic value T_H, referred to as the *Harmonizing Period*. As described in [42], the Harmonizing Period has the same initial phasing as the highest priority task τ_1, i.e, $\phi_1 = 0$. No such phasing constraints are imposed on the other tasks. In the basic *Rate-Harmonized Scheduler* (RHS), tasks that arrive before or after integral multiples of T_H are not eligible to execute until the next closest boundary of T_H, when they are serviced based on their priority [42]. For a given taskset Γ, T_H is chosen so as to improve schedulability [42]. As stated in [42], let us suppose $\Psi = \{\tau_j | T_j < 2T_1, j \neq 1\}$. If $\Psi = \emptyset$, $T_H = T_1$, otherwise $T_H = T_1/2$.

In ES-RHS, by using a periodic *Energy-Saver* task τ_{sleep} in conjunction with RHS, optimal energy savings can be achieved. The *Energy-Saver* task τ_{sleep}, is scheduled at the highest priority with its period $T_{sleep} = T_H$, initial arrival time $\phi_{sleep} = \phi_1 = 0$ and execution time $C_{sleep} \geq C_{SleepMin}$, where $C_{SleepMin}$ is a system constraint that represents the minimum round-trip time required for the processor to go into the deep-sleep state, and revert back to the active state. While using ES-RHS, the state of the processor can be broadly classified as follows [42]:

- *Busy:* The processor is executing a task $\tau_i \in \Gamma$.

[1] Task release jitter and task dependence can be incorporated using the frameworks proposed in [163] and [164], and is beyond the scope of this work.

- *Forced Sleep:* The processor is forced into *deep sleep* by the *Energy Saver* task τ_{sleep}.
- *Idle:* The processor is neither *busy* nor in *forced sleep*.

ES-RHS exhibits an interesting property, where every *idle* duration precedes, and is contiguous with the *forced-sleep* duration. Thus, idle durations can *always* be merged with the subsequent *forced-sleep* duration [42]. Hence, by *harmonizing* the executions of non-harmonic tasks, ES-RHS can yield an optimal sleep schedule. We now re-define the notion of *harmonization* to enhance the schedulability of ES-RHS.

6.1.2 Energy-Saving Rate-Harmonized Scheduling+

We first re-define the notion of *harmonization* as follows:

"A task is eligible to execute when the processor is *busy* or a *Harmonizing Period* boundary has been reached."

The above re-definition allows tasks to become eligible earlier than previously defined rate-harmonized schedulers including RHS and ES-RHS, without affecting their worst-case energy savings. Based on this new notion of harmonization, we propose ES-RHS+. To illustrate our new definition of harmonization, consider the ES-RHS schedule in Figure 6.1. The second instance of task τ_2 arrives at time $t = 23$ but only becomes eligible to execute at $t = 30$. Under our re-definition, the second instance of τ_2 becomes eligible to execute at time $t = 23$, because the processor is *busy*, i.e., the forced-sleep task is "executing". Similarly, the second instance of task τ_3 which arrives at time $t = 36$ becomes eligible at $t = 40$ under ES-RHS. Under our new definition, since the processor is busy at time $t = 36$ (executing τ_1), it becomes eligible to execute immediately. Eligible tasks will continue to be scheduled based on their respective scheduling priorities.

Using the new definition of harmonization, a task which arrives when the processor is *busy* (including *forced sleep*) becomes eligible to execute immediately. In ES-RHS, such tasks can execute only after the *next* instance of τ_{sleep} finishes execution. In the worst case, a task $\tau_j, j \neq 1$, which arrives just after the harmonizing period boundary,

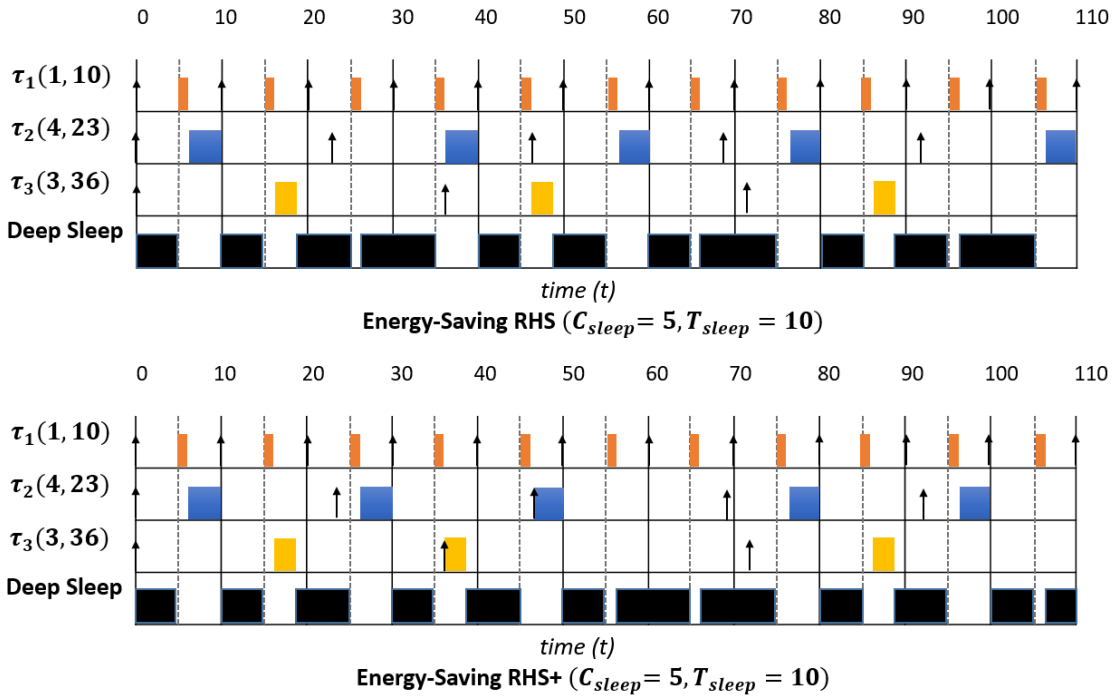


Figure 6.1: The taskset $\tau_1 = (1,10)$, $\tau_2 = (4,23)$, $\tau_3 = (3,36)$ being scheduled with ES-RHS on the top, and ES-RHS+ at the bottom. For both cases, $C_{sleep} = 5, T_{sleep} = 10$.

has to wait until the next harmonizing period boundary to become eligible to execute. This induces a worst-case blocking duration of T_{sleep}. Under ES-RHS+, the worst-case blocking for a task $\tau_j, j \neq 1$, happens when it arrives just after the Energy Saver task has *finished* execution. It becomes eligible to execute *no later* than the next harmonizing period boundary, giving rise to a worst-case blocking term of $T_{sleep} - C_{sleep}$.

We now prove that the energy savings obtained by using ES-RHS are still true for ES-RHS+.

Theorem 1: Every *idle* duration in the ES-RHS+ schedule will precede and be contiguous with a forced-sleep duration.

Proof: The Energy Saver task, τ_{sleep}, executes at the highest priority in the system, with an initial phasing $\phi_{sleep} = \phi_1 = 0$. Hence, the processor will be in forced sleep in the intervals $[(k-1)T_{sleep}, (k-1)T_{sleep} + C_{sleep})$, where, $k = 1, 2, 3, ...$ Correspondingly, the processor is considered to be *busy* in the intervals $[(k-1)T_{sleep} + C_{sleep}, kT_{sleep})$ where, $k = 1, 2, 3, ...$ Let t be any time instant at which the processor becomes *idle*, i.e., t

represents the beginning of an idle duration. Given the execution pattern followed by ES-RHS+, t must lie in the interval $[(k-1)T_{sleep} + C_{sleep}, kT_{sleep})$ where, $k = 1, 2, 3, ...$ It needs to be shown that the interval (t, kT_{sleep}), for any positive value of k, is an idle duration which in turn precedes the forced-sleep execution of the Energy Saver task τ_{sleep}.

For ES-RHS+, $T_H = T_{sleep}$. Hence, any task $\tau_i \in \Gamma$ that arrives in the interval (t, kT_{sleep}), for any positive value of k, would become eligible to execute at the next harmonizing period boundary, i.e., kT_{sleep}. Any task $\tau_i \in \Gamma$, which arrives before $(k-1)T_{sleep}$, in the worst case, would have become eligible to execute at $(k-1)T_{sleep}$. If any task arrives in the interval $[(k-1)T_{sleep} + C_{sleep}, t)$, i.e., when the processor is *busy*, then using the re-defined notion of harmonization, it would have become eligible to execute immediately. If τ_i still has some execution time left over at t, then ES-RHS+ must schedule τ_i at time t. This contradicts the assumption that t represents the beginning of an idle duration. ■

Theorem 2: Every idle duration in the ES-RHS+ schedule can be utilized to put the processor into deep sleep without any additional penalty.

Proof: From Theorem 1, we can conclude that all idle durations precede and are contiguous with the forced-sleep execution. Hence, all idle durations in the system can be combined with C_{sleep} to create a single chunked deep-sleep execution, guaranteeing that whenever the system becomes idle, it can transition into a deep-sleep duration greater than or equal to C_{sleep}. ■

Given that all idle durations in the ES-RHS+ schedule can be combined with the forced-sleep execution, the processor utilization spent in deep sleep is given by:

$$U_{sleep} = 1 - \sum_{i=1}^{n} \frac{C_i}{T_i} = 1 - U_{taskset}$$

where, $U_{taskset}$ is the total utilization of Γ. Thus, for a taskset Γ schedulable by ES-RHS+, the processor utilization spent in deep sleep is maximal. The following theorem yields the worst-case feasibility conditions, based on utilization bounds, for a taskset to be

schedulable by ES-RHS+.

Theorem 3: A taskset Γ is feasible under ES-RHS+ if

$$\frac{C_{sleep}}{T_{sleep}} + \frac{C_1}{T_1} \leq 1 \quad \wedge$$

$$\frac{C_{sleep}}{T_{sleep}} + \sum_{j=1}^{i} \frac{C_j}{T_j} + \frac{T_{sleep} - C_{sleep}}{T_i} \leq i(2^{1/i} - 1) \quad \forall i = 2 \; to \; n$$

Proof: Under ES-RHS+, it is always guaranteed that τ_1 executes immediately after the execution of τ_{sleep}. Hence, we can equivalently assume that τ_1 and τ_{sleep} together form a high-priority taskset scheduled using RMS with harmonic periods. Thus, given the harmonicity of τ_1 and τ_{sleep}, using RM-theory [29], if $(C_{sleep}/T_{sleep}) + (C_1/T_1) \leq 1$, then τ_1 is schedulable by ES-RHS+.

Consider other tasks in the taskset Γ, $\tau_i \; \forall i = 2 \; to \; n$. Compared to RMS, an instance of τ_i incurs a maximum additional delay (or blocking) of $T_{sleep} - C_{sleep}$. Hence, apart from the preemption term contributed by the execution of τ_{sleep}, the term $T_{sleep} - C_{sleep}$ can be added as a blocking term to the computational time of τ_i (C_i), and the RMS utilization bounds [29] can be used to test feasibility. ■

The tests based on RMS utilization bounds are pessimistic in nature. A more practical schedulability test utilizes the estimation of the worst-case response time of task $\tau_i \in \Gamma$. For ES-RHS+, the worst-case response time test for a task τ_i is given by the following recurrence relations:

$$W_0 = C_i + T_{sleep} - C_{sleep} \tag{6.1}$$

$$W_{k+1} = W_0 + \left\lceil \frac{W_k}{T_{sleep}} \right\rceil C_{sleep} + \sum_{j=1}^{i-1} \left\lceil \frac{W_k}{T_j} \right\rceil C_j \tag{6.2}$$

until $W_{k+1} = W_k$, in which case, W_{k+1} is the worst-case response time of the task τ_i. If $W_{k+1} \leq D_i$, then τ_i will be schedulable, otherwise τ_i will miss its deadline.

Compared to the schedulability conditions of ES-RHS [42], ES-RHS+ enhances the schedulability and feasibility conditions due to the reduction in the blocking faced by tasks. Another important property of both ES-RHS and ES-RHS+ is their inherent ability

to perform *slack stealing*. In situations where tasks do not execute up to their worst-case execution time, all the additional slack can be used to put the processor into deep sleep without any additional penalty (Theorem 2).

6.2 Energy-Saving Rate-Monotonic Scheduling

ES-RMS is a practical extension to RMS designed with the objective of maximizing energy savings in some existing operating systems. As presented in subsequent sections, ES-RMS can also help maximize energy savings for some multi-core processors. Motivated by ES-RHS, the basic Rate-Monotonic Scheduler can be extended to use a periodic *Energy Saver* task, τ_{sleep}, that executes at the highest priority with its execution time $C_{sleep} \geq C_{SleepMin}$, period $T_{sleep} = T_1$ or $T_1/2$ and phasing $\phi_{sleep} = \phi_1 = 0$. The following theorem provides the worst-case feasibility conditions for a taskset to be schedulable by ES-RMS, based on utilization bounds.

Theorem 4: A taskset on a uniprocessor is feasible under ES-RMS if

$$\frac{C_{sleep}}{T_{sleep}} + \frac{C_1}{T_1} \leq 1 \quad \wedge$$

$$\frac{C_{sleep}}{T_{sleep}} + \sum_{j=1}^{i} \frac{C_j}{T_j} \leq i(2^{1/i} - 1) \quad \forall i = 2 \; to \; n$$

Proof: Under ES-RMS, the forced-sleep execution τ_{sleep}, has an initial phasing of $\phi_{sleep} = \phi_1 = 0$ and a period $T_{sleep} = T_1 \; or \; T_1/2$. Hence, τ_1 always executes immediately after the execution of τ_{sleep}, and together form a high-priority taskset, scheduled using RMS with harmonic periods. Thus, given the harmonicity of τ_1 and τ_{sleep}, using RM-theory [29] we can say that if $(C_{sleep}/T_{sleep}) + (C_1/T_1) \leq 1$, then τ_1 is schedulable by ES-RMS.

Consider other tasks in the taskset Γ, $\tau_i \; \forall i = 2 \; to \; n$. The preemption term contributed by the execution of τ_{sleep} has to be included, and the RMS utilization bounds [29] are used to test feasibility. ■

For ES-RMS, the worst-case response time test for a task τ_i is given by the recurrence relations:

$$W_0 = C_i \tag{6.3}$$

$$W_{k+1} = C_i + \left\lceil \frac{W_k}{T_{sleep}} \right\rceil C_{sleep} + \sum_{j=1}^{i-1} \left\lceil \frac{W_k}{T_j} \right\rceil C_j \tag{6.4}$$

until $W_{k+1} = W_k$, in which case, W_{k+1} is the worst-case response time of the task τ_i. If $W_{k+1} \leq D_i$, then τ_i will be schedulable, otherwise τ_i will miss its deadline.

ES-RMS can also be readily extended to the case where task deadlines are not implicit, by using a version of *Deadline-Monotonic Scheduling* (DMS), ES-DMS.

6.3 Energy-Saving Schedulers

Both ES-RHS+ and ES-RMS (ES-DMS) are characterized by a high-priority periodic Energy-Saver task (also referred to as an ES-task or forced-sleep task). Therefore, we call this class of schedulers *Energy-Saving* Schedulers or ES Schedulers. For ES Schedulers, the generalized worst-case response time test for a task τ_i is given by the following recurrence relation:

$$W_0 = C_i, W_{k+1} = C_i + \left\lceil \frac{W_k}{T_{sleep}} \right\rceil C_{sleep} + \sum_{j=1}^{i-1} \left\lceil \frac{W_k}{T_j} \right\rceil C_j \tag{6.5}$$

where, W_{k+1} is the worst-case response time of the task τ_i. If $W_{k+1} \leq D'_i$, then τ_i will be schedulable, otherwise τ_i will miss its deadline, where, D'_i is the *generalized deadline* of a task τ_i and depends on the type of ES Scheduler used. Based on this notation, we briefly summarize each of the ES Schedulers:

(1) *ES-RMS*: Tasks execute as per rate-monotonic priorities and deadlines are assumed to be implicit ($D_i = T_i$). Here, the generalized deadline, $D'_i = T_i$.

(2) *ES-DMS*: Tasks execute as per deadline-monotonic priorities. This implies that the generalized deadline, $D'_i = D_i$.

(3) *ES-RHS+*: Tasks execute as per rate-monotonic priorities, and deadlines are implicit. However, tasks become eligible to execute based on the principle of *harmonization*:

A task is eligible to execute only when the processor is *busy* or a *Harmonizing Period* boundary has been reached [58]. The use of harmonization enables every *idle* duration in the ES-RHS+ schedule to precede and be contiguous with the ES-task. Hence, all the processor's idle durations can be utilized to put it into deep sleep, thereby providing maximal energy savings [58]. Due to harmonization, each task can be delayed by at most $T_{sleep} - C_{sleep}$ [58]. This implies that the generalized deadline, $D_i' = T_i - (T_{sleep} - C_{sleep})$, and provides a tight schedulability test compared to the slightly looser one proposed in Section 6.1.2.

6.4 Sleep-State based Energy-Saving on Multi-Core Processors

In this section, we are concerned with utilizing the processor's deep-sleep state for energy-efficient real-time multi-core scheduling. We focus on *fully-partitioned* fixed-priority scheduling. Most modern multi-core processors support a number of low-power states called C-states. In processors which have more than one C-state, individual cores can transition to idle states. However, not all processors give each core the ability to individually transition into deep sleep. Based on the ability to transition into deep sleep, we can define two types of energy-saving scheduling problems:

1) Multi-core processors where all cores can only transition *synchronously* into deep sleep. For this class of processors, we refer to the scheduling problem as "Synchronized-Sleep Multi-Core Energy-Saving Scheduling", hereafter referred to as *SyncSleep Scheduling*. Examples of such processors are Intel Core2 Duo [165] and AMD Opteron [166].

2) Multi-core processors where each core can independently transition into deep sleep. For this class of processors, we refer to the scheduling problem as "Independent-Sleep Multi-Core Energy-Saving Scheduling", hereafter referred to as *IndSleep Scheduling*. Examples of such processors are Samsung Exynos 5800 [92] and the 4th generation

Intel Core processors [167].

Consider *SyncSleep* scheduling, where the minimum amount of time for which the system can be in deep sleep is dictated by the core which has the least forced-sleep duration. To maximize energy savings, the scheduler needs to maximize the minimum forced-sleep duration across all the cores. This requires task-partitioning heuristics which efficiently distribute the load across the cores so as to achieve more energy savings. Given a balanced partition, the amount of forced-sleep duration possible on a core also depends on the scheduling algorithm used on each core.

Consider the use of ES-RHS+ on each core for *SyncSleep* scheduling. There is no guarantee that the idle durations across all cores are of the same length. Hence, all the idle time cannot be used to put the processor into deep sleep. We need to maximize and synchronize the per-core forced-sleep duration, so as to maximize the guaranteed overlap between the forced-sleep executions on each core. Thus, scheduling techniques which can guarantee a longer *synchronous forced-sleep* execution will provide greater energy savings.

Lemma 1: Consider two uniprocessor fixed-priority preemptive scheduling policies X and Y, where X has *better* schedulability conditions than Y (hence X can schedule all tasksets schedulable by Y but not vice versa). Then, given a taskset Γ and the period of the sleep task T_{sleep}, the maximum amount of time for which the processor can be in *forced sleep* for policy X is greater than or equal to policy Y.

Proof: Assume that a taskset exists, such that Y provides a longer forced-sleep duration than X. This implies that the combined utilization of the taskset and the forced sleep for Y, is greater than that for X. This contradicts the assumption that X has better schedulability conditions than Y. ■

Using Equations 6.1, 6.2, 6.3 and 6.4, we can conclude that all possible tasksets schedulable by ES-RHS+ will be schedulable by ES-RMS. However, the opposite is not true. Hence, using Lemma 1, we can say that given a taskset, ES-RMS can provide a forced-sleep duration that is greater compared to ES-RHS+. This property of ES-RMS is

beneficial for *SyncSleep* scheduling.

We extend ES-RMS and ES-RHS+ to the multi-core scheduling context by adding an Energy Saver task on each core. In the following sections, solutions for both *SyncSleep* and *IndSleep* scheduling are presented.

6.5 SyncSleep Scheduling

In this section, we formally state the energy minimization (or deep-sleep maximization) problem for *SyncSleep Scheduling*, which is as follows:

Consider a taskset Γ, consisting of n periodic real-time tasks $\tau_1, \tau_2, ..., \tau_n$ that need to be scheduled on a homogeneous multi-core processor with m cores, $M_1, M_2, ..., M_m$. Each core M_k has a Energy Saver task, $\tau_{sleep,k}$ where $k = 1, 2, ..., m$. $\tau_{sleep,k}$ has a forced-sleep duration of $C_{sleep,k}$ every $T_{sleep,k}$. The system has the constraint that all cores must synchronously transition into deep sleep. Our objective is to find a partition, and compute the global synchronized forced-sleep duration for all the cores such that:

1. The workload allocated to each core can be scheduled by ES-RMS (or ES-RHS+) in a feasible manner.

2. The synchronized forced-sleep duration is maximized, thus ensuring that the partition maximizes the minimum guaranteed energy savings among all partitions.

The above problem is a more constrained form of the feasibility problem in multi-core processor scheduling, which is known to be NP-hard in the strong sense [79]. Hence, the *SyncSleep* scheduling problem is also NP-hard. For the case, where all tasks have the same periods, with different computation time, it is similar to the Bin-Packing problem, which is known to be NP-Hard [79].

For *SyncSleep* scheduling, the forced-sleep execution must be synchronized across all the cores. Hence, for every $\tau_{sleep,k}$ where $k = 1, 2, ..., m$, we let $T_{sleep,k} = T_{sleep}$, and the initial phase be $\phi_{sleep,k} = \phi_{sleep} = 0$. Therefore, if $\min_{k=1}^{m}(C_{sleep,k}) \geq C_{SleepMin}$, then the

minimum guaranteed deep-sleep utilization is given by $\min_{k=1}^{m}(C_{sleep,k})/T_{sleep}$. For a partition to be feasible, we must have $\min_{k=1}^{m}(C_{sleep,k}) \geq C_{SleepMin}$. If $\min_{k=1}^{m}(C_{sleep,k}) < C_{SleepMin}$, then the system cannot transition into deep sleep.

Given a feasible partition and assuming that T_{sleep} is fixed for a given taskset Γ, to maximize energy savings, we need to maximize $C_{sleep,k}$ for each processing core without changing the partition, such that the tasks on each core do not become unschedulable. For each core in a feasible partition, $C_{SleepMin} \leq C_{sleep,k} \leq T_{sleep}$. By using the schedulability tests based on utilization bounds (presented in Section III and IV), we can obtain a conservative estimate of $C_{sleep,k}$ for each processing core. However, the schedulability tests based on utilization bounds are pessimistic in nature, and higher $C_{sleep,k}$ values can be achieved on each core. We now present a technique, which conducts a binary search over $[C_{SleepMin}, T_{sleep}]$ so as to efficiently compute the maximum $C_{sleep,k}$ on each core.

Consider a partitioning algorithm P, which partitions a taskset Γ onto m cores, such that the partition is feasible in the context of *SyncSleep* scheduling (either using ES-RMS or ES-RHS+). Then, for the subset of tasks $\Gamma_k \in \Gamma$ assigned to core $k \in 1, 2, ..., m$, from Equations 6.2 and 6.4, the response time of the task $\tau_i \in \Gamma_k$ is a non-decreasing function of the forced-sleep duration $C_{sleep,k}$. Since the schedulability of a task depends on its response time, for a subset of tasks $\Gamma_k \in \Gamma$ assigned to core $k \in 1, 2, ..., m$, if Γ_k is not schedulable for $C_{sleep,k} = C_x$, then Γ_k will not be schedulable for any value of $C_{sleep,k} \geq C_x$. Based on this property, we use binary search to maximize $C_{sleep,k}$ on a core k, given a partition. We call this technique *Binary "maxSleep" Search* (or BMS) and present it in Algorithm 1. BMS utilizes the response time tests (Equations 6.1-6.2 and 6.3-6.4) to test schedulability, and has a complexity of $O(log_2(T_{sleep}/\epsilon))$. Hereafter, we assume that all the partitioning schemes used in this chapter use BMS to maximize the forced-sleep execution on each core.

For a given partition, the guaranteed energy savings depends on the deep-sleep utilization [42]. As ES-RMS has better schedulability conditions than ES-RHS+, the former can provide better guarantees on minimum deep-sleep utilization for *SyncSleep* schedul-

Algorithm 1 Binary "maxSleep" Search

```
1: procedure BMS (Γ_k, C_SleepMin, T_sleep, ε)
2:     C^dn_sleep,k ← C_SleepMin                          ▷ Set lower bound
3:     C^up_sleep,k ← T_sleep                             ▷ Set upper bound
4:     TestSchedulability(Γ_k, C_SleepMin, T_sleep)
5:     if not Schedulable then
6:         return Γ_k not Schedulable
7:     while (C^up_sleep,k − C^dn_sleep,k) ≥ ε do
8:         C_sleep,k ← (C^up_sleep,k + C^dn_sleep,k)/2
9:         TestSchedulability(Γ_k, C_sleep,k, T_sleep)
10:        if Schedulable then
11:            C^dn_sleep,k ← C_sleep,k
12:        else
13:            C^up_sleep,k ← C_sleep,k
14:    return C^dn_sleep,k
   BMS
```

ing. Therefore, we consider multi-core ES-RMS, and then compare it experimentally with multi-core ES-RHS+.

6.5.1 Load Balancing to Maximize Energy Saving

Load balancing is often used to realize energy savings in multi-core systems [79]. For *SyncSleep* scheduling, it makes intuitive sense to balance the load across all cores, so as to maximize the synchronized forced-sleep duration. Based on this intuition, we state the following lemma:

Lemma 2: For *SyncSleep* scheduling, an ideal energy-aware partitioning technique is one that partitions a taskset Γ with utilization U_{total} onto m processors, such that the forced-sleep utilization on each processor is $1 - (U_{total}/m)$.

Proof: Assume an algorithm Φ exists, which for a taskset Γ, can achieve a partition with forced-sleep utilization of $1 - (U_{total}/m) + \epsilon$ on each processor, where ϵ is a finite positive quantity. Given this forced-sleep execution, the amount of processor utilization available for tasks to run is $U_{total} - m\epsilon$. This is less than the utilization required for the

taskset Γ to execute, and hence tasks will start missing their deadlines. Hence, no such algorithm Φ can exist. ■

From Lemma 2, we can conclude that, if a balanced partition exists, then an ideal algorithm creates a completely balanced partition, such that the synchronous forced-sleep utilization is maximized across all the cores in the system. Among the task-partitioning heuristics studied in the literature, the *Worst-Fit Decreasing* (WFD) algorithm is known to typically produce a well-balanced partition [79]. WFD allocates tasks one by one in non-decreasing order of their utilization. Given a task to be allocated, WFD allocates it to the core with the least utilization. For this type of problem, where WFD can allocate tasks to use only m cores, it is equivalent to *List Scheduling*, where tasks with utilization $u_i \in [0,1]$ are allocated to m cores, with the aim of reducing the maximum utilization on any core. Given that WFD and List Scheduling are equivalent in this context, we refer to both as WFD.

In [168], it was proved that, in an m-core system, for a partition generated using WFD with *Earliest-Deadline First* (EDF) scheduling (or equivalently RMS with only harmonic tasks), the core with the maximum load has a utilization of no more than $(4/3) - (1/m)$ times that of the optimal. As $m \to \infty$, the ratio becomes $4/3$. In [79], Arvind *et al.* extended this result to the case where RMS is used along with WFD. The result states that the core with the maximum load has a utilization no more than $[(4/3) - (1/m)]/ln2$ times that of the optimal. As $m \to \infty$, the ratio becomes $4/(3ln2)$. In the general case, for a scheduling technique with a per-core utilization bound U_{bound}, it can be stated that the core with the maximum load has a utilization no more than $[(4/3) - (1/m)]/U_{bound}$ times that of the optimal [79].

The approximation ratio of an algorithm is the worst-case ratio between the result obtained by the algorithm, as compared to the optimal solution. For *SyncSleep* scheduling, the approximation bound can be defined as the ratio of the synchronous forced-sleep utilization obtained using an algorithm, compared to the optimal. In the following theorems, we state and prove the approximation bound for WFD while using ES-RMS. Given

the nature of the problem, the approximation bound is a function of the synchronous forced-sleep utilization obtained by WFD. This worst-case synchronous forced-sleep utilization is the minimum forced-sleep utilization supported by the system and is given by $U_{SleepMin}$.

Theorem 5: For *SyncSleep* scheduling, the WFD heuristic for partitioning independent *harmonic* tasks onto a multi-processor, under ES-RMS, has an approximation ratio of at most:

$$\frac{4 - 3(1 - U_{SleepMin})^2}{4U_{SleepMin}}$$

Proof: Let WFD yield a partition with a synchronous forced-sleep utilization of $U_{SleepMin}$. Therefore, given that the taskset is harmonic, the combined utilization of the tasks on the core with the maximum load is $1 - U_{SleepMin}$. Let the synchronous forced-sleep utilization for the optimal partition be $U_{SleepOpt}$. Hence, the combined utilization of the tasks on the core with the maximum load is $1 - U_{SleepOpt}$. Using the result in [168], the following must hold:

$$\frac{\frac{4}{3} - \frac{1}{m}}{U_{bound}}(1 - U_{SleepOpt}) = 1 - U_{SleepMin}$$

By substituting the ES-RMS utilization bound $U_{bound} = 1 - U_{SleepMin}$, we obtain:

$$U_{SleepOpt} = 1 - \frac{(1 - U_{SleepMin})^2}{(4/3) - (1/m)}$$

The above function is maximized as $m \to \infty$. Hence, for the case where the taskset is *harmonic*, the approximation ratio for WFD under ES-RMS is:

$$\frac{U_{SleepOpt}}{U_{SleepMin}} = \frac{4 - 3(1 - U_{SleepMin})^2}{4U_{SleepMin}}$$

■

Theorem 6: For *SyncSleep* scheduling, the WFD heuristic for partitioning independent tasks onto a multi-processor, under ES-RMS, has an approximation ratio of at most:

$$\frac{4 - 3(ln2 - U_{SleepMin})^2}{4U_{SleepMin}}$$

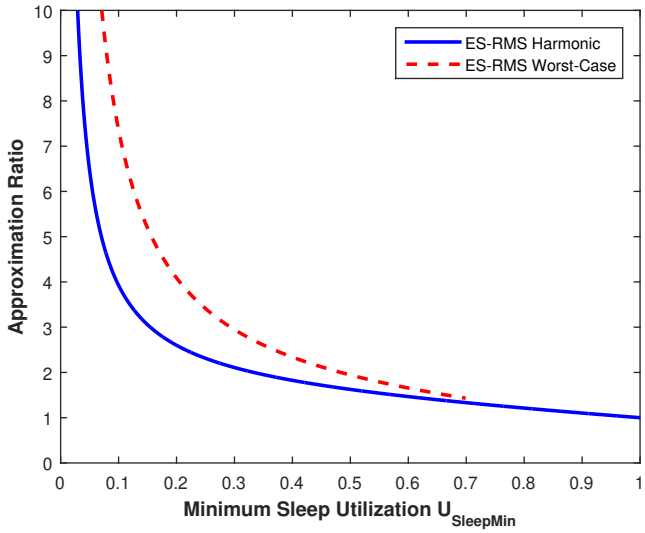


Figure 6.2: Approximation ratio for ES-RMS and WFD as a function of $U_{SleepMin}$

Proof: Let WFD yield a partition with forced-sleep utilization of $U_{SleepMin}$. For a set of n schedulable tasks the utilization bound for RMS is $n(2^{1/n} - 1)$. As $n \rightarrow \infty$, the utilization bound $U_{bound} \rightarrow ln2$. Therefore, the combined utilization of the tasks on the core with the maximum load is $ln2 - U_{SleepMin}$. Let the forced-sleep utilization for the optimal partition be $U_{SleepOpt}$. Hence, in the best case, the maximum combined utilization of the tasks on the core with the maximum load is $1 - U_{SleepOpt}$. The theorem can now be proved using the technique used to prove Theorem 5. For the sake of brevity, the details are omitted. The plot for the approximation ratios derived are illustrated in Figure 6.2. ■

Given the non-linear nature of the schedulability tests for ES-RMS (and ES-RHS+), load balancing may not always be the best approach to obtain better energy savings for *SyncSleep* scheduling. This observation is illustrated by the following example. Consider a taskset Γ which contains four tasks $\tau_1 = (40, 100)$, $\tau_2 = (40, 100)$, $\tau_3 = (105, 250)$ and $\tau_4 = (210, 500)$, which need to be scheduled on two cores M_1 and M_2. We assume that each core schedules tasks using ES-RMS. Hence, across both cores, $T_{sleep} = 50$ (here, $T_{sleep} = T_1/2$). Let us consider the following two cases:

Case 1: Tasks are assigned using WFD. τ_1 and τ_4 are assigned to M_1, and τ_2 and

τ_3 are assigned to M_2. By using the ES-RMS schedulability bounds followed by *Binary "MaxSleep" Search*, we obtain $C_{sleep,1} = 9$ and $C_{sleep,2} = 5$. Hence, for the system, the deep-sleep utilization would be $U_{sleep} = min(C_{sleep,1}, C_{sleep,2})/T_{sleep} = 5/50 = 0.1$.

Case 2: Tasks are assigned using an alternate scheme. τ_3 and τ_4 are assigned to M_1, and τ_1 and τ_2 are assigned to M_2. We now obtain $C_{sleep,1} = 8$ and $C_{sleep,2} = 10$. Hence, for the system, the deep-sleep utilization would be $U_{sleep} = min(C_{sleep,1}, C_{sleep,2})/T_{sleep} = 8/50 = 0.16$.

This example illustrates that task utilization is not the only factor which affects the time for which the system can be put into deep sleep. Other factors such as the period of a task, and how it affects the schedulability of tasks already allocated, also plays an important role.

6.5.2 Max-SyncSleep Task-Partitioning Heuristic

We now present a task-partitioning heuristic for maximizing the synchronous forced-sleep duration. Max-SyncSleep uses ES-RMS (or ES-RHS+) in conjunction with BMS, to improve the deep-sleep utilization of a multi-core processor, so as to maximize energy savings. The pseudo-code for Max-SyncSleep can be found in Algorithm 2.

Let C_{sleep} be the duration of time for which the entire multi-core processor can transition into deep-sleep state. At any instant of time, Max-SyncSleep first adds each unassigned task τ_i in the taskset Γ to each of the m cores and computes a value δ_i for each task. When a task τ_i is added to a processor k, a value δ_{ik} is computed which reflects the change in the system C_{sleep}, after adding the task. For each task τ_i, the minimum of these $\delta_{ik} \forall k \in 1, 2...m$ values is assigned to δ_i. The task having the maximum weight δ_i is chosen to be the next task to be allocated. Compared to WFD, which considers only the utilization of the task for allocation, Max-SyncSleep considers the effect of both the period and the worst-case execution time of a task on C_{sleep}. The metric δ_i effectively captures this intuition, as it measures the impact of each unassigned task on the synchronous

Algorithm 2 Max-SyncSleep Task-Partitioning Heuristic

procedure MAX-SYNCSLEEP ($\Gamma, C_{SleepMin}, m$)
  $\Gamma_{unassigned} \leftarrow \Gamma$ ▷ Unassigned tasks
  $C_{sleep} = T_{sleep}$ ▷ deep-sleep time
  for $k = 1$ to m **do**
    $\Gamma_{assigned,k} \leftarrow \phi$ ▷ Tasks assigned to core k
  while $\Gamma_{unassigned}$ is non empty **do**
    $\delta_{max} \leftarrow -\infty$
    for $i = 1$ to $Cardinality(\Gamma_{unassigned})$ **do**
      $\tau \leftarrow \Gamma_{unassigned}[i]$; $\delta_{min} \leftarrow \infty$
      for $k = 1$ to m **do**
        Add τ to $\Gamma_{assigned,k}$
        $TestSchedulability(\Gamma_{assigned,k})$
        if Schedulable **then**
          $\delta = C_{sleep} - BMS(\Gamma_{assigned,k})$
          if $\delta \leq \delta_{min}$ **then**
            $\delta_{min} \leftarrow \delta$; $\alpha \leftarrow k$
        Remove τ from $\Gamma_{assigned,k}$
      if τ not schedulable on any core **then**
        return Γ not-schedulable
      if $\delta_{min} \geq \delta_{max}$ **then**
        $task \leftarrow \tau$; $core \leftarrow \alpha$; $\delta_{max} \leftarrow \delta_{min}$
    Add $task$ to $\Gamma_{assigned,core}$
    Remove $task$ from $\Gamma_{unassigned}$
    $C_{sleep} \leftarrow Min[BMS(\Gamma_{assigned,k}) \forall k \in (1, m)]$
  return $C_{sleep}, \Gamma_{assigned,k} \forall k \in (1, m)$
Max-SyncSleep

forced-sleep duration. Once a task τ_i is selected to be assigned, we need to choose a core to allocate the task to. This allocation is done on the previously computed δ_{ik} values. The chosen τ_i is assigned to the core which provides the minimum $\delta_{ik} \forall k \in 1, 2...m$. The motivation behind this step lies in the fact that we allocate the task τ_i to a core, such that, after addition of the task, the decrease in C_{sleep} is minimized. During the execution of Max-SyncSleep, if a task is found which is not schedulable on any core, then the heuristic declares the taskset as not schedulable. The complexity of the Max-SyncSleep heuristic is $O(n^2 m log_2(T_{sleep}/\epsilon))$.

6.6 IndSleep Scheduling

For processors which enable cores to *independently* transition to deep sleep, *IndSleep* scheduling is possible, and enables greater energy savings over *SyncSleep* scheduling. On such processors, it is feasible to run ES-RHS+ on each core, so as to utilize all the idle durations to put the processor into deep sleep. We formally state the energy minimization (or deep-sleep maximization) problem as follows:

Consider a taskset Γ, consisting of n periodic real-time tasks $\tau_1, \tau_2, ..., \tau_n$ that need to be scheduled on a homogeneous multi-core processor consisting of m cores, $M_1, M_2, ..., M_m$. Each core M_k has a forced-sleep execution, $\tau_{sleep,k}$ where $k = 1, 2, ..., m$. $\tau_{sleep,k}$ has a forced-sleep duration of $C_{sleep,k}$ every $T_{sleep,k}$. Our objective is to find a partition that maximizes the sum of the forced-sleep durations on each of the cores such that the workload allocated to each core can be scheduled by ES-RHS+ in a feasible manner.

For every $\tau_{sleep,k}$, where $k = 1, 2, ..., m$, the period T_{sleep} can be chosen based on the tasks allocated to core k. While using ES-RHS+ (or ES-RHS) on each core, $C_{sleep,k} \geq C_{SleepMin}$ must hold, in order to utilize all the processor idle durations. If $C_{sleep,k} < C_{SleepMin}$, then that core cannot transition into deep sleep. Hence, for *IndSleep* scheduling using ES-RHS+, we can define a feasible partition as one in which each core has a forced-sleep execution, $C_{sleep,k} \geq C_{SleepMin}$. We now prove that using ES-RHS+ optimally solves the *IndSleep* scheduling problem.

Theorem 7: If a taskset Γ can be feasibly scheduled on m cores using ES-RHS+, with *IndSleep* scheduling, then in *any* such feasible partition, all idle durations can be used to put the processor into deep sleep.

Proof: Let the total utilization of the tasks in Γ be $U_{taskset}$. Consider a feasible partition, such that Γ is partitioned into m disjoint sets $\Gamma_k, k \in 1, 2, ..m$, where tasks $\tau_{i,k} \in \Gamma_k$ are scheduled on core k. Let the worst-case utilization of each task be $u_{i,k}$. Given that

ES-RHS+ is used on each core, the deep-sleep utilization on each core is:

$$U_{sleep,k} = 1 - \sum_{i=1}^{|\Gamma_k|} u_{i,k}$$

where, $|\Gamma_k|$ is the cardinality of the set Γ_k. Hence, the deep-sleep utilization of the system can be given by:

$$\begin{aligned} U_{sleep} &= \sum_{k=1}^{m} U_{sleep,k} = \sum_{k=1}^{m} (1 - \sum_{i=1}^{|\Gamma_k|} u_{i,k}) \\ &= m - \sum_{k=1}^{m} \sum_{i=1}^{|\Gamma_k|} u_{i,k} = m - U_{taskset} \end{aligned}$$

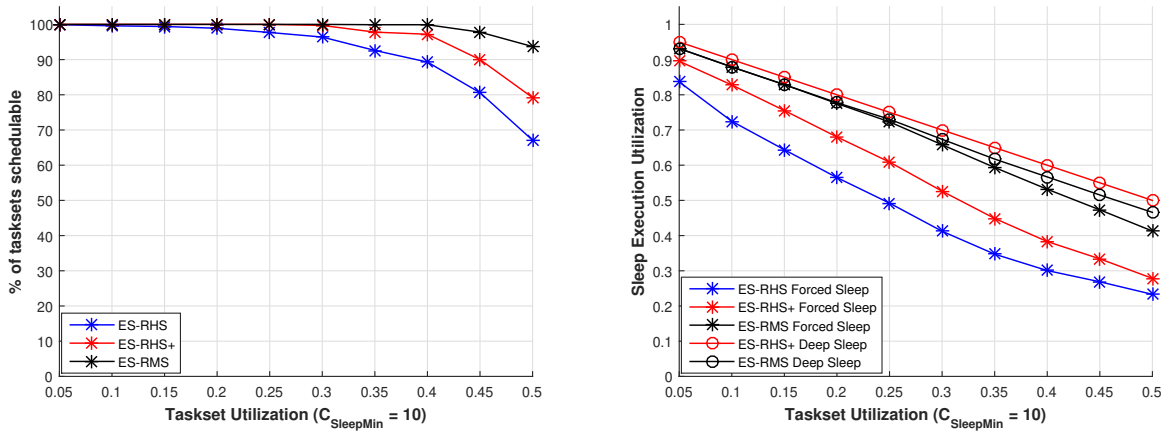
According to the theorem, using ES-RHS+ for *IndSleep* scheduling provides a lot of flexibility in task allocation, since *any* feasible partition is optimal from an energy-saving perspective. ■

Hence, we can obtain a feasible and optimal partition by using partitioning heuristics like WFD or Max-SyncSleep, by enforcing that, for each core k, $C_{sleep,k} = C_{SleepMin}$.

6.7 Comparative Evaluations

In this section, we assess the performance of ES-RHS+, ES-RMS and Max-SyncSleep. We compare different techniques on the basis of schedulability and energy savings, with respect to the total utilization of the taskset. Experiments have been performed using randomly generated tasksets. Each task is randomly assigned a period between 20 and 400 time units, and a random worst-case computation time such that the per-task utilization is always below 25%. The minimum supported forced-sleep execution, $C_{SleepMin}$ is set to 10 time units. The number of tasks used in the experiment depends on the target utilization of the taskset. Each data point we plot is an average of 1000 randomly generated task-sets with the same utilization. For each taskset, the maximum possible *forced-sleep* execution, C_{sleep} is calculated using the BMS technique.

Note: The plots in this section are best viewed in color.



(a) (b)

Figure 6.3: (a) Percentage of tasksets schedulable with respect to taskset utilization, and (b) Utilization of the forced sleep and total deep sleep with respect to taskset utilization, in the uniprocessor context

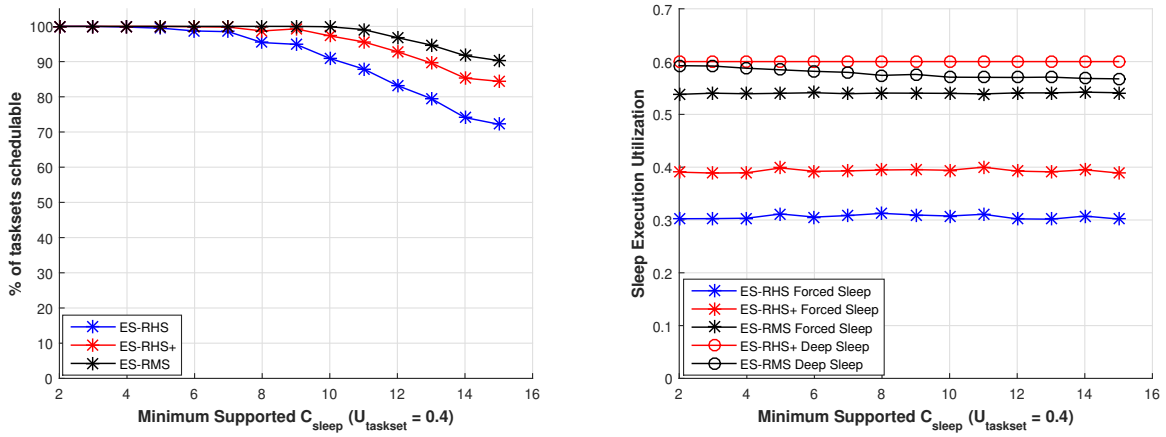
6.7.1 Uniprocessor Comparisons

In the uniprocessor context, we compare ES-RHS, ES-RHS+ and ES-RMS on the basis of schedulability, and the utilization of the forced-sleep execution $U_{sleep} = C_{sleep}/T_{sleep}$.

Figure 6.3a shows how the schedulability of the compared techniques changes as the taskset utilization is varied. In terms of schedulability; ES-RMS $>$ ES-RHS+ $>$ ES-RHS. From the plot, we can observe that ES-RMS can schedule upto 33% more tasksets than ES-RHS+.

Figure 6.3b shows how the utilization of the forced-sleep execution changes with the utilization of schedulable tasksets. Again, observe that ES-RMS always outperforms both ES-RHS+ and ES-RHS. For schedulable tasksets, ES-RMS can provide up to 18% more forced-sleep execution than ES-RHS+. This validates the use of ES-RMS for *SyncSleep* scheduling. By performing simulations over the hyperperiod of the tasksets, we also compare ES-RHS+ and ES-RMS on the basis of the total deep-sleep utilization achieved. Observe that while ES-RHS+ provides optimal energy savings, ES-RMS comes very close to the optimal.

In Figures 6.4a and 6.4b, the schedulability and sleep utilization are plotted as a



(a) (b)

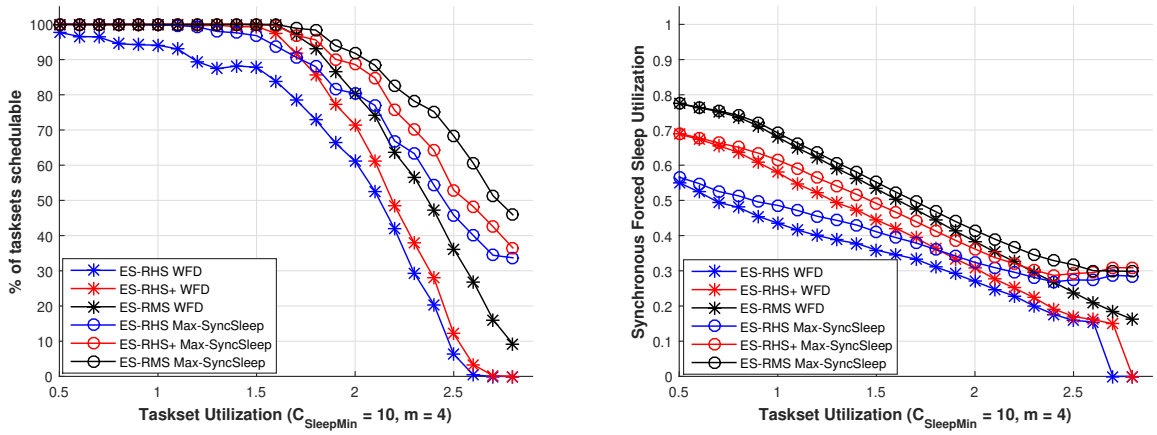
Figure 6.4: (a) Percentage of tasksets schedulable with respect to the minimum supported sleep duration, $C_{SleepMin}$, and (b) Utilization of the forced sleep and total deep sleep with respect to the minimum supported sleep duration, $C_{SleepMin}$, in the uniprocessor context

function of $C_{SleepMin}$, when the utilization of the tasksets is kept constant, $U_{taskset} = 0.4$, and $C_{SleepMin}$ is varied from 2 to 15. Observe that the forced-sleep utilization is not sensitive to $C_{SleepMin}$. However, the schedulability of the compared techniques degrades, as $C_{SleepMin}$ increases.

6.7.2 *SyncSleep* Scheduling

For multi-core *SyncSleep* scheduling, we compare ES-RHS, ES-RHS+ and ES-RMS on the basis of schedulability, and the utilization of the synchronous forced-sleep execution ($U_{sleep} = C_{sleep}/T_{sleep}$). We consider each of these techniques using both WFD and Max-SyncSleep for task partitioning.

In Figures 6.5a and 6.6a, we compare the schedulability of the techniques as a function of the taskset utilization, for a quad-core ($m = 4$) and an octa-core ($m = 8$) processor respectively. In terms of schedulability; ES-RMS $>$ ES-RHS+ $>$ ES-RHS. In the average case, combining any of these techniques with the Max-SyncSleep partitioning technique yields significantly better schedulability than WFD. In both the quad-core and octa-core cases, observe that the schedulability of the WFD-based techniques significantly

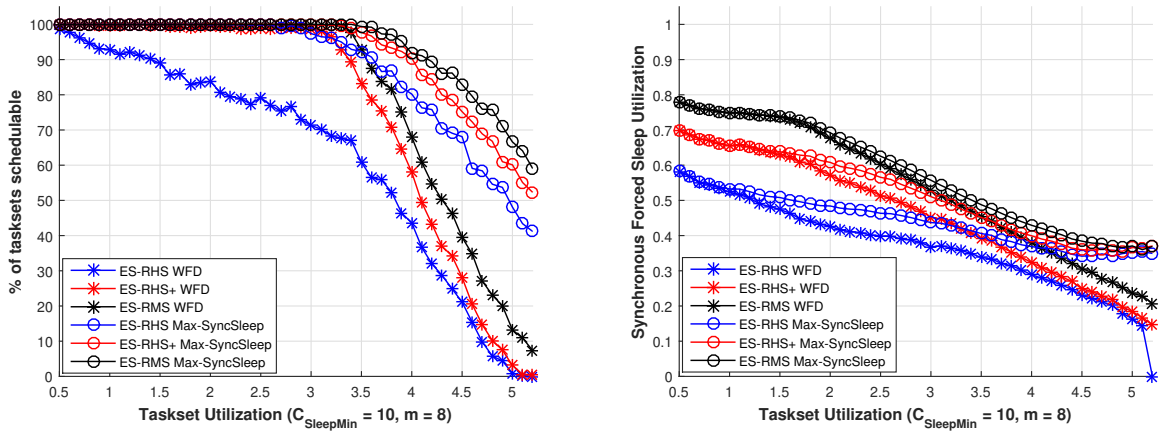


(a) (b)

Figure 6.5: (a) Percentage of tasksets schedulable with respect to taskset utilization, and (b) Utilization of the synchronous forced-sleep execution versus taskset utilization, in the multi-processor *SyncSleep* scheduling context ($m = 4$)

decreases as the number of cores/taskset utilization increases.

Figures 6.5b and 6.6b show how the utilization of the synchronous forced-sleep execution varies as a function of the utilization of schedulable tasksets for a quad-core (*m=4*) and an octa-core (*m=8*) processor respectively. Observe that ES-RMS always outperforms both ES-RHS+ and ES-RHS in this respect. Using Max-SyncSleep with ES-RMS, guarantees up to 14% more synchronous forced sleep than Max-SyncSleep with ES-RHS+, in the octa-core case. Also, note that using Max-SyncSleep provides significantly larger synchronous forced-sleep durations than WFD, thus leading to greater energy savings. The energy savings are greater, as the number of cores/utilization of the taskset increase: up to 57% more for ES-RMS with Max-SyncSleep compared to ES-RMS with WFD for the octa-core case. Hence, using Max-SyncSleep provides both better schedulability and energy savings. Note that for some taskset utilization values, using ES-RHS+ or ES-RHS with WFD does not yield any schedulable tasksets. For these cases, the obtained synchronous forced-sleep utilization is shown as zero.



(a) (b)

Figure 6.6: (a) Percentage of tasksets schedulable with respect to taskset utilization, and (b) Utilization of the synchronous forced-sleep execution versus taskset utilization, in the multi-processor *SyncSleep* scheduling context ($m = 8$)

6.7.3 IndSleep Scheduling

For *IndSleep* scheduling, we compare ES-RMS and ES-RHS+. For a feasible partition, ES-RHS+ is provably optimal in this context. Hence, we use Max-SyncSleep to generate a feasible partition for both ES-RHS+ and ES-RMS. Figure 6.7 shows the guaranteed deep-sleep utilization as a function of taskset utilization for a quad-core processor ($m = 4$). The straight line plot obtained for ES-RHS+ indicates that it optimally guarantees that all idle durations can be used to put the processor into *deep sleep*. The additional guaranteed sleep utilization obtained is up to 28% more than ES-RMS.

6.8 Summary

In this chapter, we have presented fixed-priority partitioned scheduling techniques, which utilize processor sleep states to save energy. In the uniprocessor context, we presented an enhanced version of ES-RHS. By re-defining the notion of harmonization [42], ES-RHS+ provides enhanced schedulability over ES-RHS, while still guaranteeing that all idle durations can be spent in deep-sleep state. In the multi-core context, we iden-

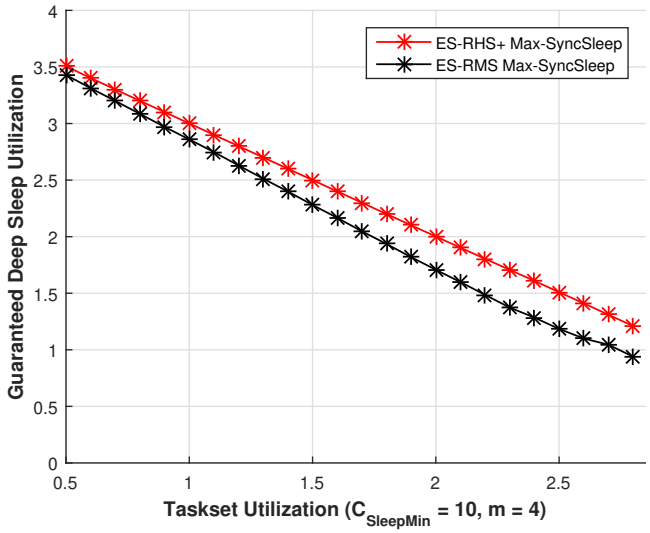


Figure 6.7: Guaranteed deep-sleep utilization versus taskset utilization, in the multiprocessor *IndSleep* scheduling context ($m = 4$)

tified two classes of scheduling problems: *SyncSleep Scheduling*, where cores need to synchronously transition to deep sleep, and *IndSleep Scheduling*, where cores can independently transition to deep sleep.

For *SyncSleep* scheduling, we proposed and used an energy-saving version of RMS called ES-RMS. ES-RMS provides greater energy savings (i.e., longer synchronous forced-sleep durations), as well as better schedulability than ES-RHS+. To maximize the synchronous forced-sleep execution, it is necessary to balance the load across cores. We proved the approximation ratio of WFD using ES-RMS as a function of the minimum possible deep-sleep utilization. We then showed that WFD does not always result in a partition with good energy savings, and proposed the Max-SyncSleep partitioning heuristic, which obtains significantly better schedulability and energy savings over WFD, by taking into account the impact of individual tasks on the synchronous forced sleep. In the *IndSleep* scheduling context, we proved that for any feasible partition, using ES-RHS+ on each core, optimally uses all the idle durations to put the processor into deep sleep.

Chapter 7

Thermal Implications of Energy-Saving Schedulers

Energy savings and system temperature are intricately tied together. As transistor geometries get smaller, the dominance of static power as a contributor to total power consumption is only expected to increase [44]. Since static power is also directly dependent on operating temperature, scheduling techniques will increasingly need to take advantage of processor sleep states.

In this chapter, we analyze the thermal properties of Energy-Saving (ES) Schedulers [58], presented in Chapter 6, which utilize the processor's deep-sleep state. The contributions described are as follows:

- We analyze the thermal performance of ES Schedulers using the well-known thermal model based on Fourier's Law, and derive design choices to pro-actively (i.e. *a priori*) minimize the maximum temperature for both uni-core and multi-core processors.
- We present the SysSleep algorithm to maximize the time the processor can be in deep sleep, and the ThermoSleep heuristic that yields a thermally-effective sleep schedule.

- We derive a lower bound on the optimal maximum temperature achievable by ES Schedulers.
- We propose task-partitioning heuristics that significantly reduce the maximum temperature for multi-core processors using ES Schedulers.
- We analyze the impact of phasing each core's forced-sleep task on temperature, in the context of multi-core processors where cores can independently transition into deep sleep.

7.1 Thermal Modeling of ES Schedulers

In this section, we introduce the thermal model used in the chapter, and derive insights in the context of ES Schedulers. The temperature of a processor is dependent on the power consumption, and the variation in power consumption over time. Therefore, we can broadly define three factors responsible for a processor's thermal profile: (i) Heat generation by a core (due to power consumption). (ii) Heat dissipation to the environment (using heat sinks). (iii) Heat dissipation between adjacent cores (due to difference in power-consumption patterns).

7.1.1 Power and Thermal Model

As described in Chapter 7, the power consumption of a CMOS circuit is modeled as a combination of two components:

(1) *Dynamic Switching Power* is dependent on the processor operating frequency, and is consumed when the processor is *busy*. The dynamic power consumption, P_D, can be modeled as a convex function of the operating frequency s as [93]: $P_D = \kappa_0 s^{\alpha}$ where, α and κ are system constants which depend on the semiconductor technology used.

(2) *Static Leakage Power* is due to leakage current, which depends on the semiconductor technology and the operating temperature. Static power is consumed even when the

processor is *idle*, but can be nearly eliminated by putting the processor into *deep sleep*. Static power, P_S, can be conservatively modeled as a linear function of temperature [93]: $P_S = \kappa_1\Theta + \kappa_2$ where, κ_1 and κ_2 are technology-dependent system constants, and Θ is the operating temperature.

Hence, the total power consumption P, as a function of time t, can be modeled as: $P(t) = P_D(t) + P_S(t)$. This model can be used to derive the thermal model for a uniprocessor. As OS schedulers control task execution at the granularity of a processor core, each core can be treated as a single unit producing heat and can be modeled as an RC circuit [93] [169]. When a core is *busy*, it generates heat. Using the RC thermal model, Fourier's Law [93] can be used to state the differential equation of the temperature, Θ^* with respect to time:

$$d\Theta^*(t)/dt = [P(t)/C] - [(\Theta^*(t) - \Theta_A)/RC] \tag{7.1}$$

where, Θ_A is the ambient temperature of the environment. By substituting P_D and P_S in Equation 7.1, we can rewrite Equation 7.1 as a classical linear differential equation [93]:

$$d\Theta(t)/dt = a - b\Theta(t) \tag{7.2}$$

where, $a = \kappa_0 s^\alpha/C$, $b = (1 - \kappa_1 R)/RC$ and the temperature has been offset from $\Theta^*(t) - [(\kappa_2 R + \Theta_A)/(1 - \kappa_1 R)]$ to $\Theta(t)$. Solving Equation 7.2 gives the temperature at time t as:

$$\Theta(t) = a/b + (\Theta(t_0) - a/b)e^{-b(t-t_0)} \tag{7.3}$$

When the processor is in deep sleep, the power consumption can be assumed to be negligible. This is a valid assumption as the difference in power consumption between the busy and deep-sleep states is different by several orders of magnitude [42]. Hence, the processor can be deemed to be cooling when in the deep-sleep state. Using this assumption, one can set $a = 0$ in Equation 7.3 to obtain the model for cooling:

$$\Theta(t) = \Theta(t_0)e^{-b(t-t_0)} \tag{7.4}$$

7.1.2 Thermal-Aware ES Scheduler Design

Consider a uni-core processor. For ES Schedulers, the processor is *guaranteed* to be in deep sleep atleast for a duration C_{sleep} every T_{sleep}. Hence, in the worst case, a core is *busy* for a duration of $T_{sleep} - C_{sleep}$ every T_{sleep}. Therefore, in the worst case, a processor core heats up from kT_{sleep} to $kT_{sleep} + C_{sleep}$ and cools down from $kT_{sleep} + C_{sleep}$ to $(k + 1)T_{sleep}$, where k is a non-negative integer. As the heating function is monotonic in the period T_{sleep}, the temperature would be maximum at the end of the heating duration. We call this temperature Θ_{max}. Similarly, as the cooling function is monotonic in the period T_{sleep}, the temperature would be minimum at the end of the cooling duration. We call this temperature Θ_{min}. Applying the heating and cooling models from Equations 7.3 and 7.4 in the duration $[kT_{sleep}, (k+1)T_{sleep})$, we can write Θ_{max} and Θ_{min} as recurrent equations:

$$\Theta^k_{max} = a/b + (\Theta^{k-1}_{min} - a/b)e^{-b(Tsleep - Csleep)}, \Theta^k_{min} = \Theta^k_{max} e^{-bC_{sleep}} \tag{7.5}$$

At steady state, as $k \to \infty$, then $\Theta^k_{min} = \Theta^{k-1}_{min}$ and $\Theta^k_{max} = \Theta^{k-1}_{max}$. Hence, the steady state worst-case values of Θ_{max} and Θ_{min} are given by:

$$\Theta_{min} = (a/b) * [(e^{bT_{sleep}(1-U_{sleep})} - 1)/(e^{bT_{sleep}} - 1)], \Theta_{max} = \Theta_{min} e^{bU_{sleep}T_{sleep}} \tag{7.6}$$

where, $U_{sleep} = C_{sleep}/T_{sleep}$ denotes the *guaranteed* utilization of the ES-task. Based on the steady state temperatures, we can draw the following conclusions:

- Increasing U_{sleep}, keeping T_{sleep} constant, decreases the maximum temperature Θ_{max}.
- Decreasing T_{sleep}, keeping U_{sleep} constant, decreases the maximum temperature Θ_{max}.

Hence, minimizing T_{sleep}, while maximizing U_{sleep}, leads to a low maximum temperature. Thus, while it is advantageous to increase the total fraction of time the processor cools, i.e. $U_{sleep} \uparrow$ (also increases guaranteed energy savings), the cooling durations

should be smaller but more frequent, i.e. $T_{sleep} \downarrow$. Note that these statements hold regardless of the system's thermal constants. Hence, using these principles, we can design techniques which can be used to minimize the temperature across a range of different systems.

In the previous chapter [42] [58], it was assumed that the period of the ES-task is a sub-harmonic of the highest-priority task. In the following section, we relax this constraint and provide techniques to design a thermally-effective ES schedule. Additionally, we show how choosing a proper T_{sleep} can maximize energy savings and improve schedulability.

7.2 SysSleep Algorithm

Consider a uni-core processor. To lower the worst-case maximum temperature for a taskset, we need to find an ES-task with a small period T_{sleep}, which also maximizes U_{sleep}. Maximizing U_{sleep} corresponds to finding the maximum *highest-priority* workload that can be added to a taskset without making it unschedulable. In [78], Saewong et al. proposed the SysClock algorithm which calculates the lowest processor frequency at which all tasks (with RM/DM priority assignment) meet their deadlines. SysClock calculates the slack at all scheduling points in the critical zone [170] to determine the optimal operating frequency. We extend that algorithm in the context of ES Schedulers, and use it to compute the set of T_{sleep} values which maximize U_{sleep}. Our algorithm is called SysSleep, and its pseudo-code is presented in Algorithm 3. We illustrate the working of SysSleep by proving its optimality.

Theorem 1. *For a taskset Γ using ES-RMS, SysSleep yields the maximum possible forced-sleep utilization U_{sleep}^{max}.*

Proof. Consider the critical zone theorem [170] where, in the worst case, the requests of all tasks arrive simultaneously. In order to be schedulable, a task τ_i must complete

before its deadline D_i, i.e., its worst-cast response time $R_i \leq D_i$. If an ES-task is added to the system, all tasks will now complete at a later time, which should still be less than D_i for the task to remain schedulable. Since the workload changes at every scheduling point, SysSleep determines the maximum workload α_i^t, that can be added to the system, such that a task τ_i completes exactly at the end of each *idle* period t between R_i and D_i. This maximum workload corresponds to the slack utilization in the schedule up to time t. While calculating α_i^t, we consider a task's execution as well as all other higher-priority tasks. For a task, the maximum workload that can be added is chosen to be the *maximum* of these candidate values. We refer to this as the *maximum additional workload*, $\rho_i^{max} = max_t(\alpha_i^t)$ for a task τ_i.

For a taskset Γ, the maximum highest-priority workload that can be added also corresponds to the maximum possible forced sleep U_{sleep}^{max}, which is the *minimum* of the *maximum additional workload* of all the tasks, i.e., $U_{sleep}^{max} = min_{\tau_i \epsilon \Gamma}(\rho_i^{max})$. Hence, U_{sleep}^{max} corresponds to the task, τ_c with the lowest *maximum additional workload*, i.e. $min_{\tau_i \epsilon \Gamma}(\rho_i^{max})$. If the added workload exceeds U_{sleep}^{max}, then τ_c will miss its deadline and the taskset will become unschedulable. □

Example: Consider a taskset Γ consisting of two tasks $\tau_1 = (1, 5)$ and $\tau_2 = (1, 7)$. For τ_1, the only end-of-idle period to consider is 5.

$$\alpha_1^5 = (t - C_1)/t = 0.8, \rho_1^{max} = max(\alpha_1^5) = 0.8$$

For τ_2, the end-of-idle periods to consider are 5 and 7.

$$\alpha_2^5 = [t - (C_1 + C_2)]/t = 0.6, \alpha_2^7 = [t - (2C_1 + C_2)]/t = 0.57, \rho_2^{max} = max(\alpha_2^5, \alpha_2^7) = 0.6$$

Hence, the maximum workload U_{sleep}^{max} that can be added is: $U_{sleep}^{max} = min(\rho_1^{max}, \rho_2^{max}) = 0.6$

We now need to find the set of T_{sleep} values which yield the maximum forced-sleep utilization U_{sleep}^{max}. For each task τ_i, let the end-of-idle period to which ρ_i^{max} corresponds be its *critical deadline*, $t_i^{critical}$. Using this notation, we can state the following lemma:

Algorithm 3 SysSleep Algorithm

procedure SysSleep(Γ)
  for $\tau_i \in \Gamma$ **do**
    $(\rho_i^{max}, t_i^{critical})$ = CalculateMaxSlack(τ_i, Γ)
  $U_{sleep}^{max} = min(\rho_i^{max}, \tau_i \in \Gamma)$ ▷ Max Sleep Utilization
  $t^{critical} = t_{argmin(\rho_i^{max})}^{critical}$ ▷ Critical Deadline
  return $U_{sleep}^{max}, t^{critical}$
procedure CalculateMaxSlack(τ_i, Γ)
  /* S = slack, I = idle duration, BusyFlag is set if core *busy*, β = workload */
  $S = I = \beta = \Delta = 0, \mu = 1$, BusyFlag=TRUE
  $\omega = C_i, \omega' = 0$
  while $\omega < D_i$ **do**
    if BusyFlag == TRUE **then** ▷ Start of a busy period
      $\Delta = D_i - \omega$
      while $\omega < D_i$ AND $\Delta > 0$ **do**
        $\omega' = \sum_{j=0}^{i} [C_j * (\lfloor \omega / T_j \rfloor + 1)] + S$ ▷ Workload Calculation
        $\Delta = \omega' - \omega, \omega = \omega'$
      BusyFlag = FALSE
    else ▷ Start of an idle period
      $I = min_{\forall j<i}[(T_j * \lceil \omega / T_j \rceil - \omega), D_i - \omega]$ ▷ Slack Computation
      $S = S + I, \omega = \omega + I, t = \omega, \beta = \omega - S$
      if $\beta / t < \mu$ **then**
        $\mu = \beta / t, t^{critical} = t, \rho = 1 - \mu$ ▷ Update the maximum additional workload
      BusyFlag = TRUE
  return $\rho, t_{critical}$

Lemma 1. *If T_{sleep} is a sub-harmonic of $t_i^{critical}$, then the ES-task τ_{sleep} can utilize all the slack ρ_i^{max} till $t_i^{critical}$, such that τ_i completes at $t_i^{critical}$.*

Proof. If T_{sleep} is a sub-harmonic of $t_i^{critical}$, the effective utilization [171] of τ_{sleep} in the duration $[0, t_i^{critical}]$ is equal to its utilization U_{sleep}. The effective utilization of a task in a duration $[0, t]$ is the fraction of processor time used by a task in that duration. The actual utilization of a task cannot exceed its effective utilization in *any* duration. Hence, τ_{sleep} can optimally utilize all the slack ρ_i^{max} in the duration $[0, t_i^{critical}]$, such that its effective and actual utilizations are equal in the duration, i.e. $U_{sleep} = \rho_i^{max}$. □

The calculated U_{sleep}^{max} corresponds to the task with the minimum ρ_i^{max}. Let us call

this the *critical task* τ_c, and let the end-of-idle period to which ρ_c^{max} corresponds be its *critical deadline*, $t_c^{critical}$. Applying Lemma 2 in the context of τ_c, we can state the following corollary:

Corollary 1.1. *If T_{sleep} is a sub-harmonic of $t_c^{critical}$, then the ES-task, τ_{sleep}, optimally utilizes all the slack, such that the critical task τ_c completes at $t_c^{critical}$.*

Unfortunately, choosing any sub-harmonic of $t_c^{critical}$ may not guarantee schedulability for other tasks in Γ. If the effective utilization of τ_{sleep} exceeds ρ_k^{max} in the duration $[0, t_k^{critical}]$, for another task $\tau_k \in \Gamma$, then τ_k will become unschedulable. Hence, we need to choose T_{sleep} such that the effective utilization of τ_{sleep} is always less than $\rho_i^{max} \forall \tau_i \in \Gamma$.

Theorem 2. *Choosing T_{sleep} as a common divisor of all $t_i^{critical} \forall \tau_i \in \Gamma$ such that $T_{sleep} \leq T_1$, always yields a schedule with the optimal forced-sleep utilization U_{sleep}^{max}.*

Proof. From Lemma 2, choosing T_{sleep} as a *common divisor* of all $t_i^{critical}$ ensures that the effective utilization U_{sleep}^{eff} of the energy-saver task τ_{sleep} is equal to its maximum utilization U_{sleep}^{max} in all the critical durations $[0, t_i^{critical}] \forall \tau_i \in \Gamma$. The optimal forced-sleep utilization is given by, $U_{sleep}^{max} = min_{\tau_i \epsilon \Gamma}(\rho_i^{max})$. Hence, $U_{sleep}^{eff} = U_{sleep}^{max} \leq \rho_i^{max} \forall \tau_i \in \Gamma$. □

It is very important to note that, in practice, the choice of T_{sleep} is constrained by the system constraint $C_{SleepMin}$ on the lower side and the period of the highest-priority task T_1 (τ_{sleep} must execute at the highest priority) on the higher side. Given this system constraint, we can state the following theorem:

Theorem 3. *Consider a taskset Γ, schedulable by an ES scheduler, running on a system with the minimum deep sleep round-trip duration $C_{SleepMin}$. Then for Γ, the lower bound on the optimal worst-case maximum temperature Θ_{max}^{best} achievable by ES schedulers is:*

$$\Theta_{max}^{best} = (a/b)[(e^{bT_{sleep}^{min}(1-U_{sleep}^{max})} - 1)/(e^{bT_{sleep}^{min}} - 1)] * e^{bU_{sleep}^{max}T_{sleep}^{min}} \tag{7.7}$$

Proof. For a taskset Γ, SysSleep returns the maximum possible forced-sleep utilization U_{sleep}^{max}. Hence, given the system constraint $C_{SleepMin}$, the smallest feasible ES-task pe-

Algorithm 4 ThermoSleep Heuristic

procedure THERMOSLEEP($\Gamma, C_{SleepMin}, num\_core$)
  while True **do**
    $U^{max}_{sleep}, t^{critical}_j$ = SysSleep(Γ) ▷ Invoke SysSleep
    if $t^{critical}_j \leq D'_j$ **then break** ▷ If critical deadline is within generalized deadline
  if $C_{SleepMin}/U^{max}_{sleep} < T_1$ **then** ▷ Check if feasible solution exists
    $\mu = \left\lfloor U^{max}_{sleep} * t^{critical}/C_{SleepMin} \right\rfloor$, $\nu = \lceil t^{critical}/T_1 \rceil$ ▷ Range of divisors
    if $\mu < \nu$ **then**
      $\mu = \nu$
    $\Theta_{best} = \infty$
    for $k = \mu$ to ν **do**
      $T^k_{sleep} = t^{critical}/k$
      Θ^{best}_k = CalcTemperature($U^{max}_{sleep}, T^k_{sleep}$) ▷ Lowest temperature for T^k_{sleep}
      if $\Theta_{best} < \Theta^{best}_k$ **then**
        break
      else
        U^{best}_{sleep} = FindSleepUtil($\Gamma, T^k_{sleep}, num\_core$) ▷ Find best U_{sleep} for T^k_{sleep}
        Θ_{max} = CalcTemperature($U^{best}_{sleep}, T^k_{sleep}$)
        if $\Theta_{max} < \Theta_{best}$ **then**
          $T_{sleep} = T^k_{sleep}, U_{sleep} = U^{best}_{sleep}, \Theta_{best} = \Theta_{max}$ ▷ Best Solution found
  else
    return NotSchedulable ▷ No feasible solution exists
  return T_{sleep}, U_{sleep}
procedure FINDSLEEPUTIL(Γ, T_{sleep}, m)
  /* m = num_cores, Γ_i = tasks allocated to core i */
  for i = 1 to m **do**
    U^i_{sleep} = FindBestSleep(Γ_i, T_{sleep}) ▷ Invoke FindBestSleep
  return $min_i(U^i_{sleep})$

riod is $T^{min}_{sleep} = C_{SleepMin}/U^{max}_{sleep}$. From Equation 7.6, the worst-case maximum temperature Θ_{max} is minimized by simultaneously minimizing T_{sleep} and maximizing U_{sleep}. Hence, substituting the smallest feasible ES-task period, T^{min}_{sleep}, and the largest schedulable forced-sleep utilization U^{max}_{sleep} in Equation 7.6 yields the lower bound on the *optimal* worst-case maximum temperature Θ^{best}_{max} achievable by ES Schedulers, corresponding to the taskset Γ. □

From a thermal perspective, for a fixed U_{sleep}, a smaller T_{sleep} yields a lower worst-

case maximum temperature. Hence, a possible thermally-effective solution with optimal forced-sleep utilization can be the smallest *common divisor* of all $t_i^{critical} \forall \tau_i \in \Gamma$ that lies in the range $[C_{SleepMin}/U_{Sleep}^{max}, T_1]$. If $C_{SleepMin}/U_{sleep}^{max} > T_1$, then no feasible solution exists. Note that, choosing T_{sleep} as *any* common divisor of $t_i^{critical} \forall \tau_i \in \Gamma$ that lies in the range $[C_{SleepMin}/U_{Sleep}^{max}, T_1]$ would yield solutions with equivalent energy consumption. However, the dependence of temperature on T_{sleep} would yield different thermal profiles.

Unfortunately, in many cases, no common divisor of the critical deadlines may lie in $[C_{SleepMin}/U_{Sleep}^{max}, T_1]$. Hence, we present the ThermoSleep heuristic. ThermoSleep invokes SysSleep to compute U_{sleep}^{max}, along with the critical deadline $t_c^{critical}$ corresponding to U_{sleep}^{max}. ThermoSleep uses these values to return the smallest possible sub-harmonic of the critical deadline $t_c^{critical}$ corresponding to the critical task τ_c, that yields a thermal and energy-efficient schedule. The pseudo-code for ThermoSleep is presented in Algorithm 4.

Given an ES-task period $T_{sleep} \leq T_1$, ThermoSleep uses the FindBestSleep (FBS) algorithm to compute the optimal C_{sleep} for a core, which allows a taskset Γ to be schedulable. The pseudo-code for FBS is provided in Algorithm 5. We now prove the optimality of FBS.

Theorem 4. *For a taskset Γ schedulable by ES-RMS, with an ES-task τ_{sleep} having a period T_{sleep}, FindBestSleep returns the optimal forced-sleep utilization U'_{sleep}.*

Proof. Consider the critical zone theorem [170] where, in the worst case, the requests of all tasks arrive simultaneously. In order to be schedulable, a task τ_i must complete before its deadline D_i. Given that a new job of τ_{sleep} is dispatched every T_{sleep}, for each task $\tau_i \in \Gamma$, FBS determines the maximum workload that can be added to the taskset, such that τ_i completes by t where, t is an integer multiple of T_{sleep}, i.e., $(k * T_{sleep} \leq D_i)$ or D_i. This gives the effective slack, α_i^t, that τ_{sleep} can utilize, if τ_i and all higher-priority tasks complete by t. For a task τ_i, the maximum highest-priority workload with period T_{sleep} that can be added is the *maximum* of these calculated values $\rho_i^{max} = max_t(\alpha_i^t)$. For a taskset Γ with an ES-task period T_{sleep}, this workload corresponds to the maximum

Algorithm 5 FindBestSleep Algorithm

procedure FINDBESTSLEEP($\Gamma, C_{SleepMin}, T_{sleep}$)
  for $\tau_i \in \Gamma$ **do**
    $(\rho_i^{max}, t_i^{critical})$ = CalculateSlack$(\tau_i, \Gamma, T_{sleep})$
  $U_{sleep} = min(\rho_i^{max}, \tau_i \in \Gamma)$ ▷ Max Sleep Utilization
  if $U_{sleep} * T_{sleep} \geq C_{SleepMin}$ **then** ▷ Check if feasible solution exists
    return $U_{sleep}^{max} * T_{sleep}$
  else
    return NotSchedulable
procedure CALCULATESLACK(τ_i, Γ, T_s)
  /* S = slack, I = idle duration, BusyFlag is set if core *busy*, β = workload */
  $S = I = \beta = \Delta = 0, \mu = 1$, BusyFlag=TRUE, $\omega = C_i, \omega' = 0$
  while $\omega < D_i$ **do**
    if BusyFlag == TRUE **then** ▷ Start of a busy period
      $\Delta = D_i - \omega$
      while $\omega < D_i$ AND $\Delta > 0$ **do**
        $\omega' = \sum_{j=0}^{i} \lceil C_j * (\lfloor \omega / T_j \rfloor + 1) \rceil + S$ ▷ Workload Calculation
        $\Delta = \omega' - \omega, \omega = \omega'$
      BusyFlag = FALSE
    else ▷ Start of an idle period
      $\Omega = \{j \in Z^+ \mid (j-1) * T_s \leq D_i < j * T_s\}$
      $I = min_{\forall j \in \Omega}[(j * T_s * \lceil \omega / j * T_s \rceil - \omega)]$ ▷ Slack computation
      $S = S + I, \omega = \omega + I, t = \omega, \beta = \omega - S$
      if $\beta / t < \mu$ **then**
        $\mu = \beta / t, \rho = 1 - \mu$ ▷ Update the maximum additional workload
      BusyFlag = TRUE
  return ρ

possible forced sleep, U'_{sleep}, which is the minimum of the ρ_i^{max} of all the tasks. Hence, $U'_{sleep} = min_{\tau_i \in \Gamma}(\rho_i^{max})$, which corresponds to the task, $\tau_c \mid c = argmin_{\tau_i \in \Gamma}(\rho_i^{max})$. If the added workload exceeds U'_{sleep}, then τ_c will miss its deadline and Γ will become unschedulable. □

For ES-RHS+, the total deep-sleep utilization $U_{SleepTotal}$ is given by $1 - \sum_{\tau_i \in \Gamma}(C_i / T_i)$. Hence, for a schedulable taskset, ES-RHS+ guarantees a sleep schedule with *optimal* energy savings. However, this deep-sleep utilization is not uniformly distributed over each period. To reduce the worst-case maximum temperature, the ES-task utilization U_{sleep} must be increased, and its period T_{sleep} must be decreased. In Section 2 the

schedulability test for ES-RHS+ was discussed, and for each task the *generalized deadline* $D'_i = T_i - (T_{sleep} - C_{sleep})$, is a function of both C_{sleep} and T_{sleep}. Hence, ThermoSleep invokes SysSleep multiple times to compute U^{max}_{sleep} until the *critical deadline* of the *critical task* lies within its *generalized deadline*. To calculate the generalized deadline, we choose T_{sleep} to be the smallest sub-harmonic of the critical deadline in the feasible range $[C_{SleepMin}/U^{max}_{sleep}, T_1]$, and $C_{sleep} = U^{max}_{sleep} * T_{sleep}$.

Given a forced-sleep period, T_{sleep}, ES-RMS can provide a higher forced-sleep utilization, U_{sleep}, than ES-RHS+ [58]. Hence, for a taskset Γ, in most cases, ES-RMS will yield a lower worst-case maximum temperature compared to ES-RHS+. In practice, ES-RHS+ can yield lower temperatures, as it utilizes *all* idle durations to put the processor into deep sleep.

7.3 Thermal-Aware Multi-Core ES Scheduling

Consider a task set Γ consisting of n periodic real-time tasks $\tau_1, \tau_2, ..., \tau_n$ that need to be scheduled on a homogeneous multi-core processor with m cores, $M_1, M_2, ..., M_m$. Each core M_k has an ES-task, $\tau_{sleep,k}$, which has a forced-sleep duration of $C_{sleep,k} \geq C_{SleepMin}$ every $T_{sleep,k}$. As mentioned in Section 2, two types of multi-core ES scheduling problems were defined in [58]. In this section, we analyze the thermal implications of *Sync-Sleep* and *Indsleep* scheduling, and propose techniques to derive thermally-effective partitioned schedules.

In multi-core processors, heat also dissipates between adjcacent cores, and the rate of dissipation depends on the temperature differences between them. Hence, each core can be modeled using the RC model with the addition of thermal resistances between adjacent cores [99]. Let the instantaneous temperature on each core be Θ_j, for $j = 1, 2, ..., m$. Using Fourier's Law, the differential equation for each core's temperature can

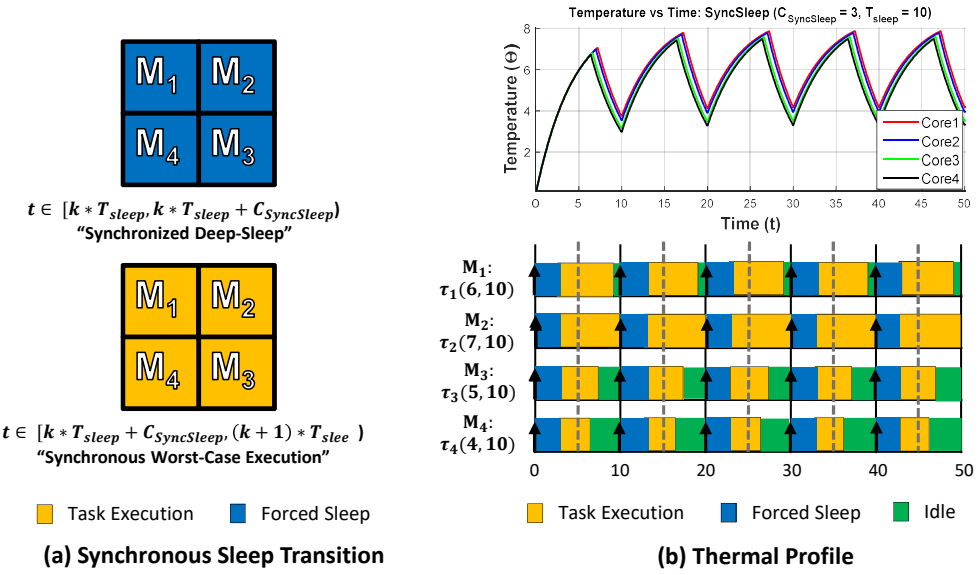


Figure 7.1: *SyncSleep* Scheduling for a quad-core system with cores M_i.

be given by:

$$\frac{d\Theta_j(t)}{dt} = \frac{P_j(t)}{C} - \frac{\Theta_j(t) - \Theta_A}{RC} - \sum_{k=1}^{m} \frac{\Theta_j(t) - \Theta_k(t)}{R_{jk}C} \tag{7.8}$$

where, P_j is the instantaneous power dissipated by the core, and R_{jk} is the thermal resistance between the cores j and k. For *non-adjacent* cores one can reasonably assume there is no heat dissipation between them and hence, $R_{jk} = \infty$ [99].

7.3.1 SyncSleep Scheduling

For *SyncSleep* scheduling, the forced-sleep task must be synchronized across all cores [58]. As the sleep transition is synchronous, for all cores $T_{sleep,k} = T_{sleep}$, and the initial ES-task phase can be taken as $\phi_{sleep,k} = 0$ [58]. Additionally, the minimum amount of time for which the system can be in deep sleep is dictated by the core which has the least forced-sleep duration [58]. Hence, if the system synchronous sleep $C_{SyncSleep} = \min_{k=1}^{m}(C_{sleep,k}) \geq C_{SleepMin}$, then the minimum guaranteed deep-sleep utilization is given by $\min_{k=1}^{m}(C_{sleep,k})/T_{sleep}$.

Based on the synchronous-sleep constraint, in the worst case, we can assume that all the cores are in deep sleep for the durations $[kT_{sleep}, kT_{sleep} + C_{SyncSleep})$, and busy from

$[kT_{sleep} + C_{SyncSleep}, (k+1)T_{sleep})$. Hence, all cores will have the same worst-case execution profile (as illustrated in Figure 7.1(a)), and we can assume that in the *worst case*, at any time instant, all cores share the *same* temperature. Thus, the worst-case inter-core temperature difference is always zero, and the model reduces to the uniprocessor thermal model. Hence, from a worst-case perspective, we can consider the entire system as one thermal unit. Applying these assumptions in Equation 7.8, the worst-case SyncSleep temperature model is given by:

$$d\Theta_j(t)/dt = P_j(t)/C - (\Theta_j(t) - \Theta_A)/RC \tag{7.9}$$

Figure 7.1(b) presents an example using SyncSleep ES-RMS for a quad-core system with cores $M_i, i = \{1,2,3,4\}$. The taskset $\Gamma = \{\tau_1(6,10), \tau_2(7,10), \tau_3(5,10), \tau_4(4,10)\}$ is used, such that, during partitioning, each core receives one task (τ_i is assigned to M_i). Due to the synchronous nature of forced sleep, all cores have similar temperature profiles, making the heat dissipation between cores negligible. Hence, like the uniprocessor case, the problem reduces to finding a forced-sleep task τ_{sleep} which minimizes T_{sleep} while maximizing U_{sleep}. However, given that we have multiple cores, partitioning the tasks among them also plays a major role in determining the thermally-effective τ_{sleep}. The temperature minimization problem can be stated as the following task-partitioning problem: "Find a partition that has a synchronized ES-task which minimizes the worst-case maximum temperature, such that the workload allocated to each core can be scheduled feasibly by an ES Scheduler."

The stated partitioning problem is a more constrained form of the feasibility problem in multi-core processor scheduling, which is known to be NP-hard in the strong sense [172] [173]. Hence, the thermal-aware *SyncSleep* scheduling problem is also NP-hard. Consider the trivial case where all tasks have the same periods, with different computation times. In this case, choosing the optimal T_{sleep} is trivial (from Theorem 4, it is a sub-harmonic of the task period). Given T_{sleep}, the temperature across all cores will be minimized if all cores have the same load. Hence, the problem reduces to cal-

Algorithm 6 SyncSleep Partitioning Heuristic

```
1: procedure PartitionTaskset(Γ, C_SleepMin, m)
2:     /* m = number of cores, Γ_i = tasks allocated to core i */
3:     T_s = 1                                          ▷ Set forced-sleep period to 1
4:     Γ_i ∀i ∈ 1 to m = MaxSyncSleep(Γ, C_SleepMin, T_s, m)   ▷ from [58]
5:     U_s, T_s = ThermoSleep(Γ, C_SleepMin, m)         ▷ Invoke ThermoSleep
6:     return U_s, T_s                                  ▷ SyncSleep task parameters
```

culating the optimal balanced partition for independent tasks with known computation times, which is known to be equivalent to the Partition problem [174] which is NP-Complete [174].

We now present a two-stage heuristic for the partitioning problem:

Partitioning for Thermal Performance: In the first stage, we choose the best possible hypothetical $T_{sleep} = 1$ to find the best synchronous forced sleep that a partitioning heuristic can achieve. Theorem 4 states that, on a single core, the optimal U_{sleep} is achieved when T_{sleep} is a common divisor of the critical deadline. Since 1 is a divisor of all integers, choosing $T_{sleep} = 1$ enables a heuristic to achieve its best possible forced-sleep utilization. If a taskset cannot be scheduled when $T_{sleep} = 1$, we consider it unschedulable. Setting $T_{sleep} = 1$ and maximizing the forced-sleep utilization is similar to the energy minimization problem for SyncSleep Scheduling [58]. To realize energy savings and minimize temperature in multi-core systems, load balancing is often used [58]. Worst-Fit Decreasing (also referred to as WFD or List Scheduling when the number of cores is fixed *a priori*) is commonly used to obtain a load-balanced partition. WFD allocates tasks to the core with the least utilization, one by one in non-increasing order of their utilization. For ES Schedulers, the period ratios also play an important role in dictating the forced-sleep utilization, something that WFD does not take into account. In [58], the *MaxSyncSleep* (MSS) partitioning heuristic was proposed. Instead of using utilization to allocate tasks to cores, MSS measures the impact of a task's allocation on the synchronous forced-sleep duration.

Choosing the SyncSleep Period: In the second stage, we find a thermally-effective

T_{sleep}. For an m-core system, let the best possible synchronous forced-sleep utilization (setting $T_{sleep} = 1$) obtained by a partitioning heuristic A be $U^{max}_{SyncSleep}$, which corresponds to the core k with the minimum forced-sleep utilization. The feasible range for T_{sleep} can now be given by $[C_{SleepMin}/U^{max}_{SyncSleep}, T_1]$. To find a good value for T_{sleep}, we run ThermoSleep on the partition. The proposed partitioning technique is described in Algorithm 6.

7.3.2 IndSleep Scheduling

Some processors allow each core to individually transition into deep sleep, enabling better energy savings. Hence, each core M_k has a forced-sleep task, $\tau_{sleep,k}$, which has a forced-sleep duration of $C_{sleep,k} \geq C_{SleepMin}$ every $T_{sleep,k}$, with a phasing $\phi_{sleep,k}$. Note that, compared to *SyncSleep* scheduling, each core's forced-sleep task can have a different $C_{sleep,k}$, as well as a different phasing $\phi_{sleep,k}$. Hence, we need to consider heat dissipation between cores. Thus, the *IndSleep* thermal model is given by Equation 7.8, and takes into account both heat dissipation to the environment, as well as between cores.

For *IndSleep* scheduling, the thermal-aware scheduling problem can be defined as follows: "Find a partition and forced-sleep task parameters (including phasing) on each core, that minimizes the maximum temperature of the system, under the constraint that the workload allocated to each core can be scheduled by an ES Scheduler."

In [58], it was proved that using ES-RHS+ can yield an energy-optimal schedule *for all* feasible partitions. A partition is feasible if the tasks allocated to each core are schedulable. However, unlike the energy-minimization problem, all the feasible partitions are *not* optimal from a thermal perspective. This is due to the dependence of temperature on the ES-task period, as well as the execution pattern between cores, i.e. relative ES-task phasing.

The heat flow between two objects is primarily dependent on their thermal properties as well as the temperature difference between them. At any instant, the temperature

difference between two adjacent cores will always be less than the temperature difference between a core and the environment. This is based on the practical assumption that the environmental temperature is *always* lower than that of any core. Thus, we can safely assume that heat dissipation to the environment is the dominant factor for cooling. Hence, from an optimization standpoint, we first optimize the schedule on each core to reduce its own temperature, and then optimize the schedule between cores to ensure maximal heat dissipation between them. Based on this practical assumption, we propose a two-stage solution:

Partitioning for Thermal Performance: The objective of partitioning is to ensure that the worst-case maximum temperature of the system is minimized. If there were no heat dissipation between cores, then the worst-case maximum temperature Θ^k_{max} on a core k is a function of $T_{sleep,k}$ and $U_{sleep,k}$. A balanced partition helps ensure that all cores have similar Θ^k_{max}. In an unbalanced partition, a core with a significantly lower $U_{sleep,k}$ would yield a higher temperature, thus raising the maximum temperature of the system. This is similar to the SyncSleep problem, and hence is also NP-Hard. Hence, like SyncSleep, we initially set $T_{sleep,k} = 1$ on each core, and use *MaxSyncSleep* [58] (or WFD) to create a balanced partition. Applying ThermoSleep to all the cores together gives a single T_{sleep} that is suitable for all the cores. We refer to this as *uniform sleep*. However, since each core can independently transition into deep sleep, each core's ES-task can have a different period, that we refer to as *non-uniform sleep*. These non-uniform sleep periods $T_{sleep,k}$ can be calculated by applying ThermoSleep to each core *individually*. FindBestSleep is then used to obtain each $C_{sleep,k}$ using the corresponding $T_{sleep,k}$. While *uniform sleep* ensures that all cores have a similar temperature profile, *non-uniform sleep* can allow each core to attain a lower temperature.

Forced-Sleep Phasing: The phasing between ES-tasks plays an important role in the heat dissipation between cores. In the worst case, we can assume that each core $M_j, j = 1$ to m is in *deep sleep* for the durations $[\phi_{sleep,j} + kT_{sleep,j}, \phi_{sleep,j} + kT_{sleep,j} + C_{sleep,j})$, and *busy* from $[\phi_{sleep,j} + kT_{sleep,j} + C_{sleep,j}, \phi_{sleep,j} + (k+1)T_{sleep,j})$. To ensure maximal heat

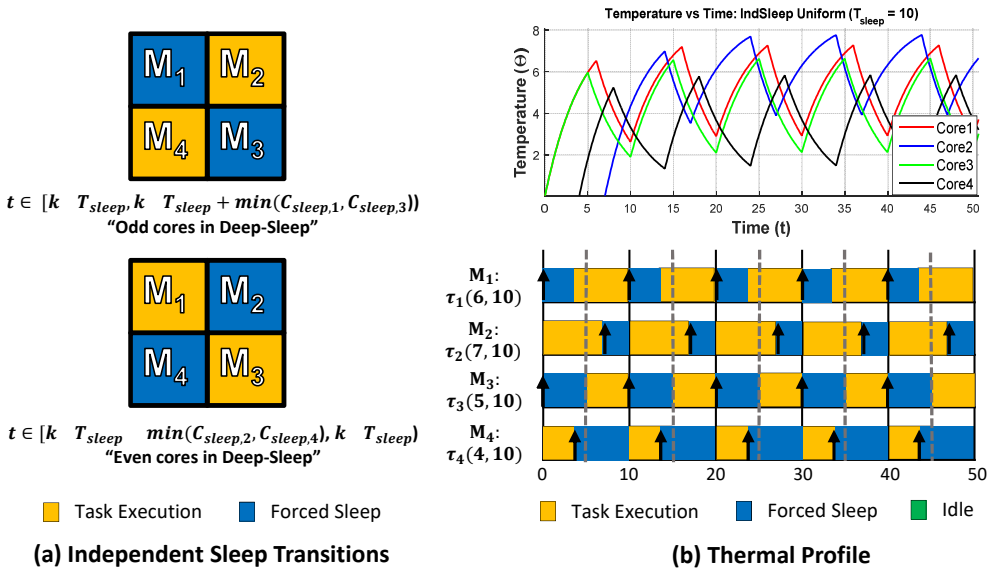


Figure 7.2: *IndSleep* Scheduling with uniform sleep periods for a quad-core system with cores M_i.

dissipation between adjacent cores, the temperature difference between them needs to be maximal. For two adjacent cores i and j, the largest temperature difference between them occurs when core i is at the start of its forced-sleep period and core j is at the end of its forced-sleep period. Hence, if $\tau_{sleep,i}$ starts exactly after $\tau_{sleep,j}$ ends, then the instantaneous temperature difference between the cores can be maximized. This leads to an execution pattern where core i is busy while core j is in deep sleep and vice versa.

Figure 7.2(b) presents an example using IndSleep ES-RMS with uniform periods for a quad-core system with cores $M_i, i = \{1, 2, 3, 4\}$. The taskset $\Gamma = \{\tau_1(6, 10), \tau_2(7, 10), \tau_3(5, 10), \tau_4(4, 10)\}$ is used, such that, during partitioning, each core receives one task (τ_i is assigned to M_i). Note that, each core has its own distinct thermal profile. Additionally, phasing the ES-task on each core, to minimize execution overlap can yield thermal benefits. For the IndSleep example, the *odd-even* execution pattern illustrated in Figure 7.2(a) is noteworthy, where execution overlap is minimized by ensuring that odd-numbered cores are *busy* (i.e. execute tasks), while even-numbered cores are in deep sleep, and vice-versa. From the thermal profile, observe that this phasing causes the temperature difference between adjacent cores to be maximized, thus yielding

better heat dissipation between adjacent cores.

As a simplification, we formulate the phasing problem as one of: "minimizing the execution (or forced-sleep) overlap between adjacent cores". By considering busy durations as *hot* and forced-sleep durations as *cool*, the execution overlap metric captures the durations where *hot* regions overlap, hence acting as a proxy for temperature difference. In most processor designs, cores are rectilinear, and adjacent cores are of the same size. Hence, to compute a thermally-effective phasing, the overlap between every pair of adjacent cores needs to be minimized. This execution overlap (also referred to as *overlap*) needs to be calculated over the *relative hyperperiod*, T_R, of all the cores. We define the *relative hyperperiod* as the least common multiple of all the cores' forced-sleep periods. In the simplest case, consider a dual-core system, with two adjacent cores. Let the cores be M_1 and M_2, and their forced-sleep tasks be $\tau_{sleep,i} = (C_{sleep,i}, T_{sleep,i})$ with phasing $\phi_{sleep,i}$ where, $i = 1, 2$. Assume that all the terms are integers, which is reasonable as we can convert timescales to arbitrarily small units (like nanoseconds). We have four possible cases:

1) $T_{sleep,1} = T_{sleep,2}$, i.e. uniform sleep. The phasing with the minimum overlap is computed over $T_R = T_{sleep,1} = T_{sleep,2}$. The minimum overlap possible is $T_R - C_{sleep,1} - C_{sleep,2}$. Then, $\phi_{sleep,1} = 0$, $\phi_{sleep,2} = C_{sleep,1}$, is one of the phasings which *guarantees* minimum overlap.

2) $T_{sleep,1}$ and $T_{sleep,2}$ are *relatively prime*, i.e. non-uniform sleep whose greatest common divisor is 1. The minimum overlap needs to be computed over $T_R = T_{sleep,1} * T_{sleep,2}$. In this case, any relative *integer phasing* of $\tau_{sleep,1}$ and $\tau_{sleep,2}$ guarantees the same overlap, which is the minimum overlap. This stems from the fact that all possible relative integer phasings between two periods are encountered, before the relative phasing is equal to that at the start.

3) $T_{sleep,1}$ and $T_{sleep,2}$ are *harmonic*, i.e. non-uniform sleep where one is a multiple of the other. Let $T_{sleep,2} = a * T_{sleep,1}, a \in Z^+$. Hence, $T_R = T_{sleep,2}$, and only one iteration of $\tau_{sleep,2}$ occurs in T_R. Then $\phi_{sleep,1} = 0$, $\phi_{sleep,2} = C_{sleep,1}$ can *guarantee* the minimum

overlap.

4) $T_{sleep,1}$ and $T_{sleep,2}$ are not *relatively prime* and not *harmonic*, i.e. non-uniform sleep which share a common divisor, but one is non-divisible by the other. Here, $T_R < T_{sleep,1} * T_{sleep,2}$. In this case, no property can be stated on the relative phasing which guarantees minimum overlap.

Based on the above properties, we see that while simple approaches work for phasing uniform sleep, using non-uniform sleep requires more complex optimization techniques.

However, using *uniform sleep* does not always guarantee lower execution overlap than using *non-uniform sleep*. This can be seen from the following 3 cases:

Case 1: *Uniform Sleep* performs better than *Non-Uniform Sleep*. Consider a taskset with two tasks, $\tau_1 = (6, 9)$ and $\tau_2 = (10, 15)$. τ_1 is assigned to core M_1, and τ_2 to core M_2. In the *uniform sleep* case the best ES-task periods in terms of sleep utilization are $\tau_{sleep,1} = (3, 9)$ and $\tau_{sleep,2} = (2.5, 9)$. Using the best possible phasing, achieves a guaranteed minimum execution overlap of 3.5 every 9 (38.89%). In the non-uniform case, the best ES-task periods are $\tau_{sleep,1} = (3, 9)$ and $\tau_{sleep,2} = (5, 15)$. By searching the entire search space of unique relative integer phasings, the minimum execution overlap achievable is 20 every 45 (44.44%). Hence, in this case, using *uniform sleep* provides lower execution overlap.

Case 2: *Uniform Sleep* performs equal to *Non-Uniform Sleep*. Consider a taskset with two tasks, $\tau_1 = (6, 9)$ and $\tau_2 = (9, 12)$. τ_1 is assigned to core M_1, and τ_2 to core M_2. In the *uniform sleep* case, the best ES-task periods in terms of sleep utilization are $\tau_{sleep,1} = (3, 9)$ and $\tau_{sleep,2} = (1.5, 9)$. By using the best phasing, we can achieve a guaranteed minimum execution overlap of 4.5 every 9 (50%). In the non-uniform case, the best ES-task periods are $\tau_{sleep,1} = (3, 9)$ and $\tau_{sleep,2} = (3, 12)$. By searching the entire search space of unique relative integer phasings, the minimum execution overlap achievable is 18 every 36 (50%). Hence, both provide a solution with the same execution overlap.

Case 3: *Uniform Sleep* performs worse than *Non-Uniform Sleep*. Consider a taskset with two tasks, $\tau_1 = (6, 9)$ and $\tau_2 = (9, 11)$. τ_1 is assigned to core M_1, and τ_2 to core M_2.

In the *uniform sleep* case, the best ES-task periods are $\tau_{sleep,1} = (3, 9)$ and $\tau_{sleep,2} = (1, 9)$. Using the best phasing, achieves a guaranteed minimum execution overlap of 5 every 9 (55.55%). In the non-uniform case, the best ES-task periods are $\tau_{sleep,1} = (3, 9)$ and $\tau_{sleep,2} = (2, 11)$. By searching the entire search space of unique relative phasings, the minimum execution overlap achievable is 54 every 99 (54.54%). Hence, in this case using *non-uniform sleep* provides lower execution overlap.

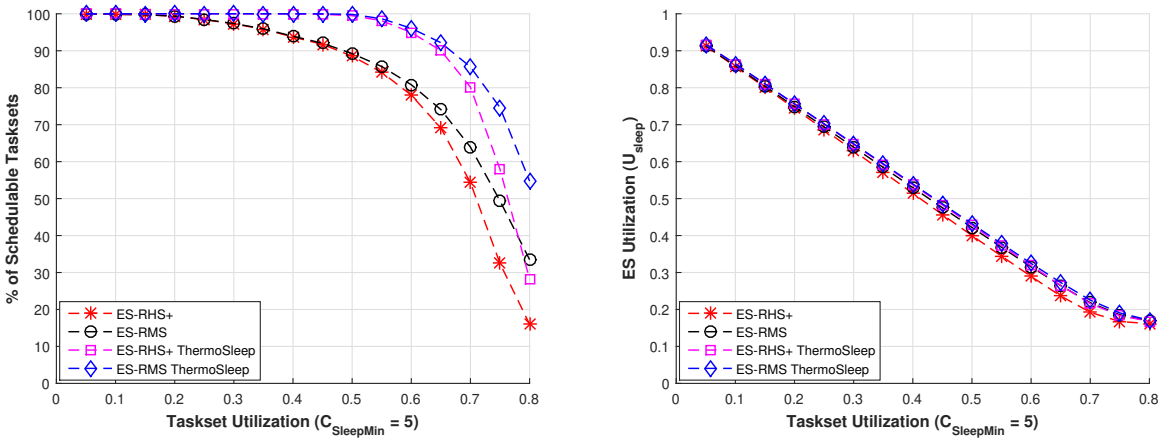
Since there is no exact solution for choosing ES-task periods for minimizing execution overlap, we examine the properties of using *uniform sleep* versus *non-uniform sleep*:

Best Phasing: While uniform sleep can be phased easily and optimally (using the odd-even execution pattern from Figure 7.2(a)) for a rectilinear multi-core processor, no such simple technique can be used for non-uniform sleep.

Temperature Profile: Uniform sleep will ensure that all cores have similar temperatures. However, using non-uniform sleep allows each individual core to choose the best T_{sleep}, to further reduce its temperature, based on the tasks allocated to it.

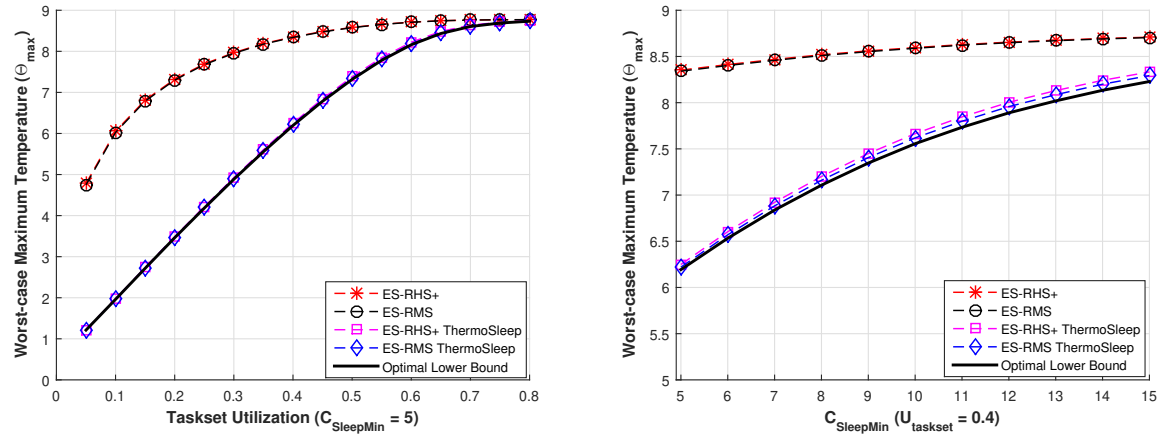
7.4 Comparative Evaluation

We now evaluate our proposed techniques on the basis of schedulability and worst-case maximum temperature Θ_{max} with an offset. Results are obtained using both static worst-case analysis as well as dynamic simulations using Hotspot [169]. Static analysis experiments were performed on 100,000 tasksets generated randomly using UUniFast-Discard [175] for each data-point. In a taskset, each task is randomly assigned a period between 15 and 400 time units, and the number of tasks varies from 1 to 20. $C_{SleepMin}$ is set to 5 time units. The system thermal parameters were set to $a = 2$ and $b = 0.228$ [93]. To the best of our knowledge, no other *proactive* techniques exist for designing thermal-aware fixed-priority sleep schedules. Hence, we compare against the purely energy-efficient design methodology proposed in the previous chapter [58].



(a) (b)

Figure 7.3: Uniprocessor Results (a) Utilization of the ES-task w.r.t taskset utilization, and (b) % of task sets schedulable w.r.t taskset utilization

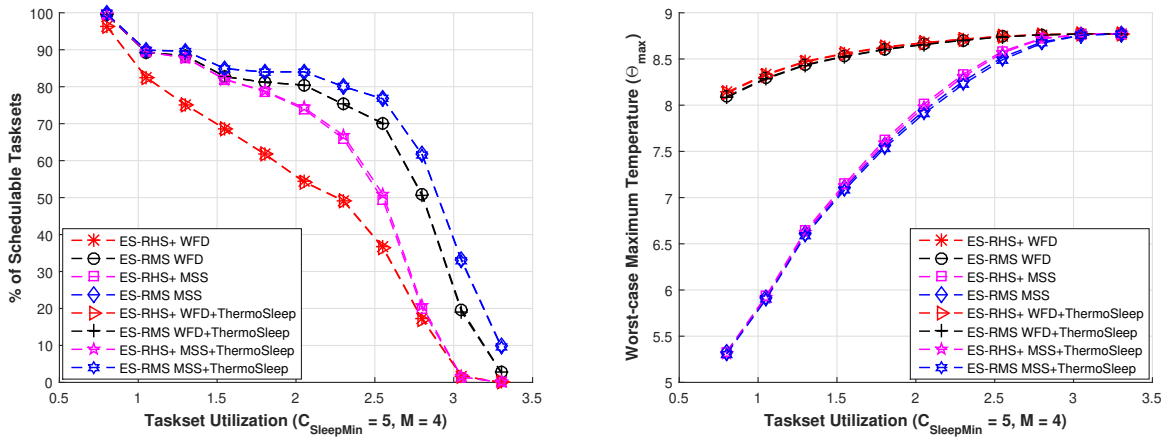


(a) (b)

Figure 7.4: Uniprocessor Results (a) Worst-case maximum temperature w.r.t taskset utilization, and (b) Worst-case maximum temperature w.r.t $C_{SleepMin}$

7.4.1 Static Worst-Case Analysis

Uniprocessor Comparisons: We compare ES-RMS and ES-RHS+ with and without using ThermoSleep on the basis of schedulability, and the worst-case maximum temperature, Θ_{max}. Figure 7.3a plots schedulability versus taskset utilization. In terms of schedulability: ES-RMS performs better than ES-RHS+. Observe that, using ThermoSleep, ES-RMS can schedule up to 62.5% more task sets than before. Figure 7.3b plots the ES-task utilization versus taskset utilization for tasksets schedulable by all techniques. By using



(a) (b)

Figure 7.5: SyncSleep Results (a) % of task sets schedulable w.r.t taskset utilization, for multi-core *SyncSleep* scheduling ($m = 4$), and (b) Worst-case maximum temperature w.r.t taskset utilization, for multi-core *SyncSleep* scheduling ($m = 4$)

SysSleep, ThermoSleep-based techniques yield slightly greater ES-task utilization —up to 3.3% greater for ES-RMS. Figure 7.4a plots Θ_{max} versus taskset utilization for tasksets schedulable by all techniques. Despite the ES-task utilization being similar, by choosing a smaller ES-task period, ThermoSleep can achieve significantly lower temperatures —on average up to 4°K lower for ES-RMS, while simultaneously yielding better energy savings. Figure 7.4a also plots the average of the lower bound on Θ_{max} for the tasksets. On average, the worst-case deviation between the solution provided by ES-RMS and ThermoSleep, and the optimal lower bound was 0.028°K. Figure 7.4b plots Θ_{max} as a function of $C_{SleepMin}$, when taskset utilization $U_{taskset} = 0.4$. Observe that, despite varying $C_{SleepMin}$, our approach yields solutions with a worst-case temperature difference of 0.067°K compared to the optimal lower bound.

Multi-core SyncSleep Comparisons: We compare ES-RMS and ES-RHS+ on the basis of schedulability and the worst-case maximum temperature, Θ_{max}. We consider each technique using both WFD and *Max-SyncSleep* (MSS) for task partitioning, with and without using ThermoSleep. For a quad-core ($m = 4$) processor, Figure 7.5a plots schedulability versus taskset utilization, and Figure 7.6a plots the utilization of the synchro-

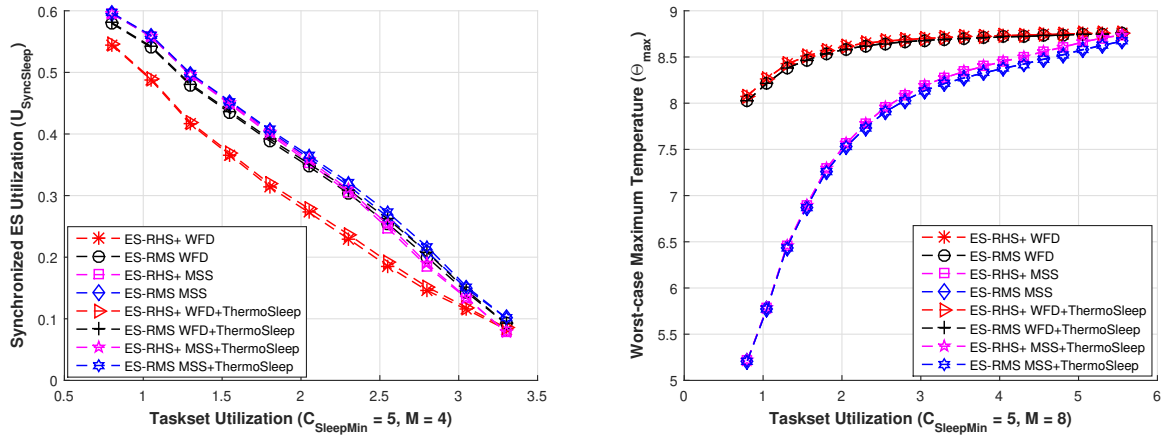
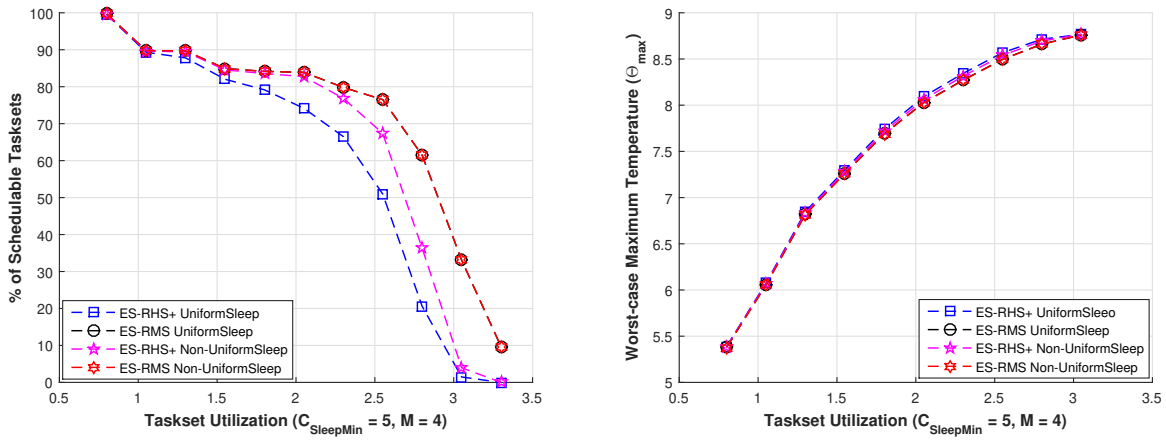


Figure 7.6: SyncSleep Results (a) Synchronized ES-task utilization w.r.t taskset utilization, for multi-core *SyncSleep* scheduling ($m = 4$), and (b) Worst-case maximum temperature w.r.t taskset utilization, for multi-core *SyncSleep* scheduling ($m = 8$)

nized ES-task $U_{SyncSleep}$. In terms of schedulability and $U_{SyncSleep}$ ES-RMS performs better than ES-RHS+ for all partitioning techniques. For partitioning techniques, MSS marginally dominates WFD in terms of schedulability and $U_{SyncSleep}$, both with and without using ThermoSleep. Using ThermoSleep provides marginally better $U_{SyncSleep}$ —up to 11.59% greater for ES-RMS with MSS. Figures 7.5b and 7.6b plot Θ_{max}, versus taskset utilization, for a quad-core ($m = 4$), and an octa-core ($m = 8$) processor respectively. Using ThermoSleep can give significantly lower Θ_{max} —on average up to 2.89°K lower for ES-RMS with MSS for $m = 4$.

Multi-core IndSleep Comparisons: We compare ES-RMS and ES-RHS+ using *MaxSyncSleep* to generate partitions. We consider using both uniform and non-uniform sleep for each core's ES-task. MSS along with ThermoSleep is used to determine the sleep periods. Figure 7.7a plots the percentage of schedulable tasksets. Note that using non-uniform sleep allows for greater schedulability —up to 1.2% greater for ES-RMS. Figure 7.7b plots Θ_{max} without considering inter-core heat dissipation. Note that, while ES-RHS+ provides maximal energy savings, in all cases ES-RMS yields lower temperatures than ES-RHS+. Additionally, due to better use of each core's *idle* durations, non-uniform



(a) (b)

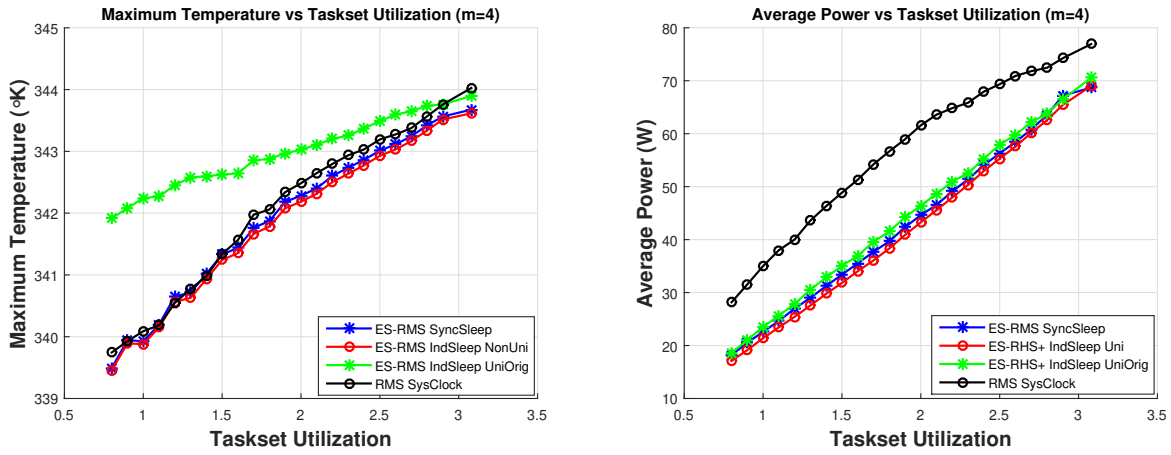
Figure 7.7: IndSleep Results (a) % of task sets schedulable w.r.t taskset utilization, for multi-core *IndSleep* scheduling ($m = 4$), and (b) Worst-case maximum temperature w.r.t taskset utilization, for multi-core *IndSleep* scheduling ($m = 4$)

sleep provides slightly lower temperatures than uniform sleep —up to 0.08°K.

7.4.2 Dynamic Simulations

To perform dynamic thermal simulation, we have designed a real-time multi-core scheduling simulation tool called *Inferno* (v1.0). Based on the processor floor-plan, prior temperature, power consumption in the interval and the interval length, *Inferno* uses Hotspot [169] to calculate each core's temperature, in each scheduler-simulation interval. *Inferno* supports fully-partitioned fixed-priority scheduling. Simulation parameters such as the number of cores, simulation cycles, simulation granularity, $C_{SleepMin}$, floor-plan and thermal configuration can be specified by the user. The power consumption of each core for different operating frequencies in the *busy*, *idle* and *deep-sleep* states are specified in a look-up table. Based on the taskset and partition provided by the user, *Inferno* provides a trace of the power and temperature values at each simulation instant. The source code for *Inferno* can be found at *https://github.com/sandeepdsouza93/Inferno*.

In order to use realistic power values, we considered the automotive benchmark from the MiBench suite [176]. The benchmark was compiled and executed in the SniperSim



(a) (b)

Figure 7.8: Dynamic simulations (a) maximum temperature w.r.t taskset utilization ($m = 4$), and (b) average power w.r.t taskset utilization ($m = 4$)

[177] cycle-accurate x86 emulator (for a Nehalem-class x86 processor) for a range of frequency settings (1.22-2.66 GHz). The execution trace obtained from SniperSim is then fed to the McPAT [178] power simulator, which calculates the power consumption based on an x86 Nehalem power model (45 nm technology node). To model the dependency of static power on temperature, McPAT power calculations were done for the range of temperatures: 300-400°K, and the values were stored in a look-up table. *Inferno* uses these values to compute the core power consumption value, based on the previously calculated core temperature. The scheduling simulation granularity was set to 10μs, and Hotspot's default thermal configuration was used.

We have simulated a quad-core processor, with the floor-plan consisting of cores laid out in a square grid (as shown in Figures 7.1(a) and 7.2(a)). 10,000 randomly generated tasksets were considered, each containing 1 to 20 tasks. The taskset utilization varied from 0.8 to 3.2. Each taskset was simulated up to thermal steady state (Hotspot warm-up was considered).

Figure 7.8a plots maximum temperature versus taskset utilization. Observe that ES-RMS *IndSleep* with non-uniform sleep yields the lowest temperature. We also compared techniques from [58] following a purely energy-efficient design (UniOrig), and

it returned the highest temperature —on average up to 3.91°K of difference between ES-RMS IndSleep without thermal considerations (UniOrig), and ES-RMS IndSleep with non-uniform sleep. We also compare our techniques with SysClock, which is the energy-optimal fixed-priority technique for static frequency scaling. We simulate SysClock with RMS where each core could have its own frequency. SysClock yields higher temperatures than IndSleep —up to 1.5°K higher.

Figure 7.8b plots the average power consumption versus taskset utilization. We find that by better utilizing the idle durations, ES-RHS+ IndSleep yields lower power consumption than ES-RMS SyncSleep —up to 5.04 W lower. ES-RHS+ IndSleep on average yields a power consumption that is 8.52 W lower than SysClock with a maximum difference of 21.74 W. This highlights the importance of energy-saving techniques based on sleep states. Our techniques also provide greater power savings compared to the purely energy-efficient design methodology presented in [58] —up to 8.36 W additional power-savings for ES-RHS+ IndSleep. Note that, although ES-RHS+ IndSleep provides greater energy savings, ES-RMS IndSleep yields lower temperatures. Additionally, even though SysClock consumes significantly more power than ES Schedulers, they both yield similar maximum temperatures. This highlights the fact that energy efficiency does not always imply lower temperatures.

7.5 Summary

In this chapter, we analyzed the thermal implications of fixed-priority energy-saving schedulers, which utilize the processor's deep-sleep state to save energy. We infer design choices from a well-known thermal model, and presented two techniques for designing thermally-effective ES Schedulers: the SysSleep algorithm to provide optimal sleep utilization and the ThermoSleep heuristic to design a thermally-effective ES-task. Specifically, we derive a lower bound on the optimal maximum temperature, thus quantifying the best thermal performance achievable by ES Schedulers. In the multi-core

context, we extend our analysis to two classes of scheduling problems [58]: *SyncSleep*, where cores need to synchronously transition into deep sleep, and *IndSleep*, where cores can independently transition into deep sleep. We consider the impact of both task partitioning and ES-task phasing on temperature. In the SyncSleep context, we observe that the synchronous deep-sleep constraint reduces the temperature-minimization problem to the energy-minimization problem, with the exception of the synchronous ES-task period calculation. On the other hand, while energy minimization is straightforward in the IndSleep context (all feasible partitions are optimal using ES-RHS+ [58]), the same cannot be said for temperature minimization. The dependence of temperature on the ES-task periods and relative phasing makes the IndSleep problem non-trivial.

Since we focus on fully-partitioned scheduling, our proposed framework can be extended to heterogeneous multi-core processors. Additionally, our techniques do not require significant knowledge of a system's thermal parameters, and hence are applicable to a range of multi-core platforms. Static analysis and dynamic simulation validate our approach, yielding lower temperatures and better energy savings than both the purely energy-efficient ES Scheduler design [58], and frequency scaling based techniques [78]. Our results show that, while energy savings is key to lower temperatures, not all energy-efficient solutions yield low temperatures.

Chapter 8

Energy-Saving Scheduling for Real-Time Systems with Hardware Accelerators

Hardware accelerators often consume significant amounts of power [46]. In addition, energy-constrained platforms such as smartphones, drones, robots and AR/VR headsets also contain one or more hardware accelerators [20] [179] [180]. Hence, it is necessary to focus on energy management for systems using hardware accelerators.

Like multi-core processors, hardware accelerators can also expose power-management interfaces. However, in commercial accelerators like GP-GPUs and DSPs, only P-states are exposed to the user [47] [48]. Thus, in effect, they expose only voltage and frequency-scaling knobs for power management, and the job of reducing static power is done in firmware or hardware. Therefore, we focus on using frequency-scaling-based techniques to reduce the power consumption of systems using hardware accelerators. In particular, we propose techniques to statically set the processor and accelerator to a pre-computed taskset-specific frequency, such that the aggregate energy consumption is reduced, while ensuring that all deadlines are met. The use of static frequency-scaling techniques involves setting the processor and/or the accelerator to a pre-computed taskset-specific frequency. Therefore, as there are no dynamic frequency changes, the unpredictable latency involved in changing the oscillator frequency is avoided, leading

to more deterministic operation, which is desirable in real-time systems.

The primary contributions described in this chapter are as follows:

- We introduce a novel search technique called *ratchet search*, and use it to jointly estimate the upper and lower bounds of the range containing the lowest feasible frequency, as additional tasks and resources are considered
- We propose the CycleSolo family of algorithms to calculate the energy-optimal frequency-scaling factor, when (i) the frequency of only the CPU or the accelerator can be scaled, or (ii) both the CPU and accelerator frequency must be scaled by the same common factor.
- We propose the CycleTandem algorithm to calculate low-power frequency-scaling factors for the CPU and the accelerator, when both the CPU and accelerator frequency can be independently scaled.
- We extend the CycleSolo and CycleTandem algorithms to the fully-partitioned multi-core context.

8.1 System Model

In this section, we present the assumptions and system model used in this chapter. We also provide some background about the synchronization-based approach used to govern access to hardware accelerators [52], along with its suspension-based schedulability analysis introduced in prior work [53].

8.1.1 Assumptions and Task Model

Consider a taskset Γ consisting of n sporadic real-time tasks $\tau_1, \tau_2, ..., \tau_n$. The taskset is deployed on an m-core homogeneous multi-core processor M, with a single non-preemptive hardware accelerator A. This assumption is reasonable as most existing

accelerators, including GP-GPUs and DSPs, do not support preemption [53]. We model the accelerator as a shared resource, and assume that access to the accelerator is treated as a *critical section* arbitrated by a global lock L [52]. We also assume that, at any point of time, only a single task can utilize the accelerator. This avoids any unpredictability in execution time caused by accelerators which support concurrent execution [53]. Additionally, accelerators used in energy-constrained platforms may not support concurrent execution.

Based on the above assumptions, each task $\tau_i \in \Gamma$ is characterized by $\{C_i, G_i, T_i, D_i\}$, where C_i is the worst-case execution time (WCET) on the CPU, G_i is the WCET on the accelerator, T_i is the period or minimum job inter-arrival time (sporadic tasks), and D_i is the relative deadline from the arrival time. The term G_i consists of: (i) G_i^e, the WCET of the task on the accelerator, and (ii) G_i^m, the worst-case CPU-intervention required to access the accelerator. Note that $G_i \leq G_i^e + G_i^m$, as G_i^e and G_i^m may not occur on the same control path [53]. However, we assume that $G_i = G_i^e + G_i^m$. Therefore, for each task τ_i, we define the total CPU time required as $E_i = C_i + G_i^m$. We also assume that each task can access the accelerator at most once every period. This assumption is reasonable since, in practice, most tasks have a single accelerator-executed segment. Tasks which have multiple accelerator segments can be split into separate tasks. We also assume that, while accessing the accelerator, each task suspends on the CPU.

In this chapter, we consider fully-partitioned fixed-priority preemptive scheduling and assume deadlines are constrained, i.e., $D_i \leq T_i$. Task priorities are assumed to be unique with each task τ_i assigned the priority π_i. The taskset is listed in decreasing order of task priorities, i.e., $\pi_1 > \pi_2 > ... > \pi_n$.

MPCP-based Analysis: In the context of this work, we utilize the Multiprocessor Priority Ceiling Protocol (MPCP) [105] to govern access to the global lock L used to access the accelerator [53]. We consider the version of MPCP, where a task requesting access to a critical section locked by another task is suspended and inserted into a lock-specific priority queue [53]. When a task releases a lock, the task at the head of the

priority queue is scheduled, and granted access to the critical section. In doing so, the priority of the task is raised to the lock's priority ceiling. The priority ceiling of the critical section of task τ_i accessing lock L, is given by $\pi_i = \pi_k + \pi_B$, where π_B is a priority level greater than the base priority of any task in the system and π_k is the highest base priority of any task that uses L. On completion of the critical section, τ_i releases the lock, and its priority is returned to its original base priority.

To determine taskset schedulability, we use response-time-based analysis. Based on this technique, the worst-case response time for a task τ_i is given by the following recurrence:

$$W_i^0 = C_i + G_i + B_i, W_i^{k+1} = C_i + G_i + B_i + \sum_{h=1}^{i-1} I_{i,h} \tag{8.1}$$

where, W_i is the worst-case response time of the task τ_i, B_i provides an upper-bound on the worst-case blocking faced by τ_i in getting access to the accelerator, and $I_{i,h}$ denotes the worst-case CPU preemption τ_i faces due to a higher-priority task τ_h. If $W_i \leq D_i$, then τ_i will be schedulable.

The worst-case preemption $I_{i,h}$, faced by task τ_i due to a higher-priority task τ_h is given by:

$$I_{i,h} = \alpha_{i,h} * E_h, \alpha_{i,h} = \lceil (W_i + W_h - E_h)/T_h \rceil \tag{8.2}$$

where, $\alpha_{i,h}$ represents an upper bound on the number of jobs of τ_h released during a single job of τ_i [53]. Note that $\alpha_{i,j}$ considers the jitter, $W_h - E_h$, introduced by τ_h's self-suspension on the CPU, while accessing the accelerator [107].

The worst-case blocking B_i, faced by a task τ_i, in accessing the accelerator can be upper-bounded by multiple approaches described in prior work [53] [54] [55]. Three such approaches are (i) the job-driven analysis, (ii) the request-driven analysis, and (iii) the hybrid analysis. Neither of the first two analyses strictly dominates the other. In practice, the work in [53] observed that the job-driven analysis dominates the request-driven analysis when the number of critical sections a task executes on the accelerator increases. On the other hand, as C_i and W_i increase the job-driven analysis becomes

more pessimistic. The hybrid analysis proposed in [53] uses a combination of the job-driven and request-driven analysis to provide a less-pessimistic worst-case response-time estimate.

In Section 8.1.1, we assumed that each task has only one critical section which is executed on the accelerator. Under this assumption, we can prove the following theorem.

Theorem 1: If all tasks in Γ have *at most* one critical section executed on the accelerator, then the request-driven analysis *always* dominates the job-driven analysis.

Proof: Consider the case where all tasks in Γ have at most one critical section executed on the accelerator. Both the request-driven and job-driven analyses determine schedulability using response-time calculations (Equation 8.1). The only difference lies in the calculation of *high-priority* blocking faced by each task τ_i. In the request-driven analysis, the number of instances of each higher-priority task τ_h, which contribute to blocking τ_i, is given by the term $\beta_{i,h}$, which upper-bounds the number of accelerator requests made by a higher-priority task τ_h, while τ_i is being blocked. Thus, $\beta_{i,h} = \left\lceil \frac{B_i + W_h - E_h}{T_h} \right\rceil$, where B_i upper bounds the time for which τ_i is blocked [53] [55]. In contrast, for the job-driven analysis, the number of instances of each high-priority task τ_h, which contribute to blocking τ_i, is given by the term $\alpha_{i,h}$, which upper-bounds the number of accelerator requests made by a higher-priority task τ_h, during τ_i's response time. Thus, $\alpha_{i,h} = \left\lceil \frac{W_i + W_h - E_h}{T_h} \right\rceil$, where W_i upper-bounds τ_i's worst-case response time [53]. From Equation 8.1, for every task τ_i, we can conclude that $B_i < W_i$. Therefore, for every feasible pair of τ_i and τ_h, $\beta_{i,h} \leq \alpha_{i,h}$. This implies that, for the given context, the request-driven analysis always computes a tighter blocking estimate, as compared to the job-driven analysis. Thus, from Equation 8.1, we can also conclude that the request-driven analysis always computes a tighter worst-case response-time estimate, as compared to the job-driven analysis. ■

As a corollary, to the above theorem, it can also be proven that the request-driven analysis is equivalent to the hybrid analysis [53]. Therefore, in this work, we utilize the request-driven analysis. However, the algorithms we propose can easily be adapted to other analysis techniques.

Based on the request-driven analysis [53] [55], the worst-case blocking B_i, faced by a task τ_i in accessing the accelerator, can be upper-bounded by the following recurrence [53]:

$$B_i^0 = \max_{\tau_l | l>i}(G_l), B_i^{k+1} = \max_{\tau_l | l>i}(G_l) + \sum_{h=1}^{i-1} \beta_{i,h} * G_h \tag{8.3}$$

where, $\beta_{i,h} = \lceil (B_i + W_h - E_h)/T_h \rceil$ upper bounds the number of accelerator requests made by a higher-priority task τ_h, while τ_i is being blocked. Note that $\beta_{i,h}$ considers the self-suspension of a task on the CPU, while accessing the accelerator.

8.1.2 Power Model

The power consumption of modern CMOS-based processors is modeled as a combination of two major components:

(1) *Dynamic Power* is dependent on the processor operating frequency. Assuming that voltage is scaled with frequency, dynamic power consumption, P_D, can be modeled as a convex function of the operating frequency s as [83]: $P_D = Kf^{\alpha}$ where, α and K are technology-dependent system constants.

(2) *Static Power* is due to leakage current, which depends on the semiconductor technology. Static power, P_S, can be modeled as [83]: $P_S = VI_{leak}$ where, V is the operating voltage and I_{leak} is the technology-dependent leakage current.

Hence, power consumption $P = P_D + P_S$. Therefore, the total power consumed by the CPU-accelerator combination can be given by $P_{total} = P_{cpu} + P_{acc}$, where P_{cpu} and P_{acc} are the power consumption of the CPU and accelerator respectively.

While dynamic power is reduced using voltage and frequency scaling, static power is reduced using sleep states. However, as mentioned earlier, most accelerators do not provide user-configurable sleep states to reduce static power [181], and rely on firmware-based control to reduce static power. Therefore, we focus on using frequency-scaling-based power management, and assume that the processor/accelerator performs its own optimizations in parallel to reduce static power. In particular, we focus on statically

choosing a single operating frequency for the CPU/accelerator. This is based on the well-known property that, for processors with a *non-decreasing* convex power-frequency function, the energy is minimized if the processor executes its workload at the lowest-possible constant frequency [78].

For the sake of simplicity, we assume a continuous processor/accelerator frequency range normalized to the range $[0, 1]$. In later sections, we discuss how discrete frequencies can be accommodated. We also assume that task worst-case execution times (WCET) are specified at the maximum frequency, $f_{max} = 1$, and the WCET is scaled in proportion to the operating frequency f, i.e, $WCET_f = WCET_{f_{max}}/f$. However, our proposed algorithms are independent of this scaling model, and any model where the execution-time monotonically increases with decreasing frequency can be used.

8.2 CycleSolo Algorithm

We now introduce the CycleSolo family of algorithms for uniprocessor systems with a single hardware accelerator. We propose three variants of CycleSolo:

1) CycleSolo-CPU: when the accelerator does not support frequency scaling, and only the CPU frequency can be scaled.

2) CycleSolo-Accel: when the accelerator supports frequency scaling, but the CPU frequency cannot be scaled.

3) CycleSolo-ID: when both the accelerator and the CPU frequencies must be scaled by a common scaling factor.

The two steps of the CycleSolo family of algorithms are as follows: 1) Compute a tight bound on the frequency range in which the optimal frequency lies. 2) Perform a binary search testing schedulability over the computed frequency range, to obtain the lowest-possible CPU/accelerator operating frequency which ensures that all deadlines are met.

Algorithm 7 Binary Search Minimizing Frequency

```
1: procedure BinarySearch(Γ, ε_conv, f_high, f_low)
2:     while f_high − f_low > ε_conv do
3:         f_est = (f_high + f_low)/2                ▷ chosen frequency
4:         Γ′ = ScaleTaskset-Frequency(Γ, f_est)
5:         if Γ′ is Schedulable then f_high = f_est
6:         else f_low = f_est
7:     return f_high
```

We can easily prove that given a schedulability-analysis technique, a binary search (Algorithm 7) will always converge to the lowest-possible operating frequency.

Lemma 1: Given a taskset Γ, and a response-time-based schedulability-analysis technique S, a binary search testing schedulability over the operating frequency range, converges to the lowest-possible operating frequency f_{min}, which guarantees that Γ is schedulable using technique S.

Proof: Consider an operating frequency $0 < f \leq f_{max}$. For every task τ_i, the worst-case execution time is inversely proportional to the frequency. Therefore, the response time of a task also increases monotonically as the frequency decreases. Thus, we can conclude that the taskset will be schedulable for all frequencies $f' \geq f$, if and only if the taskset is schedulable at frequency f, using technique S. Conversely, if a taskset is not schedulable at frequency f, it will not be schedulable for all frequencies $f' \leq f$. Given that Γ transitions from schedulable to unschedulable after frequency f_{min}, the estimated frequency range decreases with each iteration of the binary search (Algorithm 7), and, given sufficient iterations, converges to $f_c = f_{min}$, which is the lowest frequency guaranteeing Γ is schedulable according to analysis technique S. ■

However, performing the response-time-based schedulability test multiple times over the entire frequency range $[0, f_{max}]$ is not desirable, as the response-time-based analysis has pseudo-polynomial complexity. Therefore, we now explain how CycleSolo computes a tight bound on the range in which the optimal frequency lies, by proving various results in the context of CycleSolo-CPU. However, the same results can be easily extended

to the entire CycleSolo family of algorithms.

CycleSolo-CPU: Consider a uniprocessor with a non-preemptive accelerator, whose frequency is not adjustable. Therefore, to minimize energy, we need to find the lowest CPU operating frequency at which all tasks meet their deadlines.

In [78], the SysClock algorithm was proposed for independent sporadic tasks using fixed-priority uniprocessor scheduling. SysClock calculates the lowest processor frequency at which all tasks meet their deadlines. For each task τ_i, SysClock calculates the slack at all scheduling points in the critical zone [170] to determine the minimum frequency, f_i, at which τ_i meets its deadline, in the presence of high-priority interference. SysClock finally chooses the lowest frequency f_{min}, as the maximum of these per-task minimum frequencies, i.e., $f_{min} = \max_{i|\tau_i \in \Gamma} f_i$. Thus, SysClock chooses the lowest frequency at which all tasks meet their deadlines.

In this work, we use the slack-calculation methodology from SysClock. However, unlike SysClock, the following issues are encountered while estimating the minimum frequency:

1) **Undefined Critical Instant**: In the presence of blocking and self suspensions, the critical instant *does not* necessarily occur when all high-priority jobs arrive together with the task τ_i, and is undefined [107]. Therefore, additional blocking and self-suspension terms are added to utilize the existing response-time analysis. In effect, this assumes the same critical instant, but adds extra pessimism by considering the worst-case blocking and modeling the self-suspensions as release jitter [107]. Therefore, like SysClock, the CycleSolo algorithms consider the critical-zone theorem [170] where, in the worst case, the requests of all tasks arrive simultaneously. In practice, the worst-case blocking, interference and self-suspension penalties never appear together. Due to this pessimism, all the known analysis techniques are safe, but none of them are exact [53]. This prevents us from finding the absolute minimum frequency. Instead, we can obtain the minimum frequency which allows a taskset to be schedulable *given* the analysis framework used.

2) **Frequency-Dependent Slack**: Due to self suspensions, the interference and block-

ing faced by each low-priority task depends on the response time and worst-case execution time of higher-priority tasks. However, the worst-case response time of higher-priority tasks depends on the operating frequency. Thus, different operating frequencies create different amounts of high-priority interference and blocking, which makes the slack calculation frequency-dependent. In SysClock, it is sufficient to calculate the slack at each scheduling point, i.e., a task's deadline or points in time when a new job of a task is released. However, in the presence of self-suspending tasks, the pessimism added to the response-time analysis changes the *effective* points in time at which new instances of tasks appear, and makes them dependent on the frequency.

Lemma 2: Consider response-time-based schedulability-analysis techniques which model self-suspension as release jitter. Then, for a task τ_h which self-suspends, the set of effective scheduling points, i.e., points in time where new instances of a task *effectively* arrive are given by:

$$S_h := \{j * T_h - (W_h - E_h) | j > 0\} \tag{8.4}$$

Proof: The definition of the response-time analysis stated in Equation 8.1, calculates the number of jobs, $\alpha_{i,h}$, of each higher-priority task τ_h which interfere with the execution of a low-priority job τ_i. For a higher-priority task τ_h, which self-suspends on the CPU, at each time instant $t \in S_h$, the interference increases by the execution time of one job of τ_h. Therefore, they can be considered as effective scheduling points in the context of the response-time analysis being used. ■

Lemma 2 indicates that the interference calculation depends on the high-priority response time, which depends on the operating frequency. This dependence does not occur for tasks which do not self-suspend, as is the case in SysClock, and prevents us from calculating the lowest frequency in a single pass over all the tasks. We instead estimate a feasible range $[f^{low}, f^{high}]$ which contains the lowest frequency, f_{min}.

Algorithm 8 presents the pseudo-code for CycleSolo-CPU, which considers tasks in decreasing order of their priority. For each task $\tau_i \in \Gamma$, CycleSolo-CPU computes a range

$[f_i^{low}, f_i^{high}]$, which contains the lowest frequency ensuring that τ_i and all higher-priority tasks $\tau_{h|h<i}$ are schedulable.

Consider a task $\tau_i \in \Gamma$. Let us assume that the range $[f_{i-1}^{low}, f_{i-1}^{high}]$ has already been computed, and is known. To calculate the minimum frequency at which τ_i and all higher-priority tasks are schedulable, we need to estimate the available slack in the schedule. Based on Lemma 2, for a task τ_i, the workload β_i^t changes at every scheduling point $t \in \{S_h \mid h < i, t \leq D_i\} \cup D_i$. Therefore, CycleSolo-CPU determines the CPU workload β_i^t, that exists in the system up to each scheduling point t. However, for CycleSolo-CPU, the scheduling points depend on the frequency at which the high-priority workload is run. In particular, as the frequency decreases, the response time of high-priority tasks $W_{h|h<i}$ is monotonically non-decreasing. Thus, the estimated workload β_i^t also depends on the frequency f_h *chosen* for the higher-priority tasks, and is monotonically non-decreasing as the frequency f_h decreases. Assuming we choose a frequency f_h, CycleSolo-CPU's slack calculation would yield a frequency $f_i^{est} = \min_{t \in S_h, D_i} \beta_i^t / t$, where, $h < i$ and $t \leq D_i$.

We need to choose the frequency f_h, such that the obtained f_i^{est} can provide a safe range $[f_i^{low}, f_i^{high}]$ containing the lowest frequency f_i^{min} which can ensure that τ_i and all higher-priority tasks $\tau_{h|h<i}$ are schedulable.

Lemma 3: For a task $\tau_i \in \Gamma$, choosing the high-priority frequency f_h as f_{i-1}^{high}, yields a *correct* range $[f_i^{low}, f_i^{high}]$ in which lies the lowest frequency f_i^{min} guaranteeing that τ_i and all higher-priority tasks are schedulable using analysis S.

Proof: Let us choose $f_h = f_{i-1}^{high} - \delta$, where $\delta \in (0^+, f_{i-1}^{high} - f_{i-1}^{low}]$. Now, in the worst case, the minimum frequency required to schedule only the higher-priority tasks $\tau_{h|h<i}$ can be $f_{i-1}^{min} > f_h$. Therefore, at frequency f_h, at least one of the higher-priority tasks $\tau_{h|h<i}$ will miss their deadlines. As the slack calculation used to estimate f_i^{est} uses the higher-priority worst-case response time W_h, this deadline violation will lead to an incorrect estimate of f_i^{est}. Therefore, there is a contradiction. Thus, $\delta = 0$, which implies

Algorithm 8 Minimizing CPU Frequency

procedure CycleSolo-CPU(Γ, ϵ_{conv})
  $f_{low} = f_{high} = U_{cpu}$ ▷ initial bounds from Lemma 4
  for $\tau_i \in \Gamma$ **do** ▷ from high to low priority
    f_{low}, f_{high} = EstimateFreqRange($\tau_i, \Gamma, f_{high}, f_{low}$)
  f_{min} = BinarySearch($\Gamma, \epsilon_{conv}, f_{high}, f_{low}$)
  return f_{min}
procedure EstimateFreqRange($\tau_i, \Gamma, f_{high}, f_{low}$)
  $f_h = f_{high}$ ▷ Lemma 3, f_h= high-priority frequency
  /* S = slack, ω = resp time, β = CPU workload */
  $S = I = \beta = \Delta = 0, f_{est} = 1$, BusyFlag=TRUE
  W = Calculate-HP-ResponseTime(f_h)
  B_i = CalculateBlocking(τ_i, Γ, f_h, W)
  $\omega_g = G_i + B_i - G_{cpu}$ ▷ Accelerator execution time
  $\omega = C_i + G_i + B_i, \omega' = 0, J_i = \omega$
  while $\omega < D_i$ **do**
    if BusyFlag == TRUE **then**
      $\Delta = D_i - \omega$
      while $\omega < D_i$ AND $\Delta > 0$ **do**
        $\omega' = \sum_{h=0}^{i-1} E_h * [\left\lfloor \frac{\omega + W_h - (E_h/f_h)}{T_h} \right\rfloor + 1]$
        $\omega' = \omega' + J_i + S, \Delta = \omega' - \omega, \omega = \omega'$
      BusyFlag = FALSE
    else ▷ Start of an idle period
      t = Find-EarliestSchedulingPoint(τ_i, Γ, ω)
      $S = S + (t - \omega), \omega = t, t' = \omega - \omega_g, \beta = \omega - S - \omega_g$
      if $\beta / t' < f_{est}$ **then**
        $f_{est} = \beta / t'$
      BusyFlag = TRUE
  f_{low}, f_{high} = RatchetSearch-Step($f_{high}, f_{low}, f_{est}$)
  return f_{low}, f_{high}
procedure CalculateBlocking(τ_i, Γ, f_h, W)
  $G_{l,max} = max_{\tau_l \in lp(\tau_i)}(G_l)$, $B = G_{l,max}$, $B' = 0$
  while B != B' **do**
    $B' = B$, $B = G_{l,max} + \sum_{\tau_h \in hp(\tau_i)} \lceil \frac{B' + W_h - \frac{E_h}{f_h}}{T_h} \rceil * G_h$
  return B

$f_h = f_{i-1}^{high}$. ■

Ratchet Search: A "ratchet search" is an incremental technique that refines earlier estimates based on additional parameters. Algorithm 9 presents the RatchetSearch-Step

Algorithm 9 Calculating the CycleSolo Frequency Bounds

1: **procedure** RATCHETSEARCH-STEP($f_{high}, f_{low}, f_{est}$)
2: /*f_{high} = upper bound, f_{low} = lower bound*/
3: **if** $f_{est} > f_{high}$ **then** ▷ check estimate against bounds
4: $f_{low} = f_{high}$; $f_{high} = f_{est}$ ▷ Case III
5: **elseif** $f_{est} > f_{low}$
6: $f_{low} = f_{est}$ ▷ Case II
7: **return** f_{low}, f_{high} ▷ chosen frequency range

routine, which is a single step of RatchetSearch, in the context of estimating the bounds of the frequency range $[f_i^{low}, f_i^{high}]$.

Consider the frequency f_i^{est} obtained by choosing the high-priority frequency $f_h = f_{i-1}^{high}$. We can have three scenarios:

Case I. $f_i^{est} < f_{i-1}^{low}$ implies that $\tau_h \in \Gamma \mid h \leq i$ will be schedulable at f_h, as the available slack is sufficient to support an operating frequency $f_i^{est} \leq f_h$. Hence, no change is needed to the existing frequency range, as $f_i^{min} \in [f_{i-1}^{low}, f_{i-1}^{high}]$.

Case II. $f_i^{est} \in [f_{i-1}^{low}, f_{i-1}^{high}]$. Of all the frequencies in the previously-computed feasible range, choosing $f_h = f_{i-1}^{high}$, introduces the *least*-possible high-priority interference. This creates the *maximum*-possible slack for τ_i, and enables f_i^{est} to be minimized for the frequencies in the feasible range. Therefore, the minimum frequency f_i^{min} required to schedule tasks $\tau_h \in \Gamma \mid h \leq i$ is always greater than f_i^{est}. Thus, the lower bound of the range f_i^{low} can be safely updated to f_i^{est}.

Case III. $f_i^{est} > f_{i-1}^{high}$ implies that τ_i is not schedulable at $f_h = f_{i-1}^{high}$. Therefore, the lower bound of the range f_i^{low} can be safely updated to f_h. The previous statement also implies that task τ_i would not be schedulable for any frequency $f < f_h$. Therefore, choosing $f_h = f_{i-1}^{high}$ also introduces the *maximum*-feasible high-priority interference, which minimizes the available slack and allows f_i^{est} to be maximized. Thus, f_i^{est} is a safe and tight upper bound which guarantees that tasks $\tau_h \in \Gamma \mid h \leq i$ are schedulable. Therefore, the upper bound of the range f_i^{high} can be safely updated to f_i^{est}.

However, the RatchetSearch routine requires an initial estimate of the upper and lower

bounds of the range, f_{init}^{low} and f_{init}^{high}, in which the minimum frequency lies. As RatchetSearch always increases or "ratchets up" the value of the bounds, both bounds can be initialized to $f_{init}^{low} = f_{init}^{high} = 0$.

Lemma 4: The lowest-possible CPU frequency, f_{min}, at which a taskset Γ is schedulable is always greater than or equal to $U_{cpu} * f_{max}$, where U_{cpu} is the CPU utilization of the taskset at the maximum operating frequency f_{max}.

Proof: Let the taskset Γ be schedulable at frequency $f' = U_{cpu} * f_{max} - \epsilon$, for some $\epsilon > 0$. At this frequency f', the CPU utilization of the taskset will be $U_{cpu} * f_{max}/f' > 1$. Therefore, the taskset cannot be schedulable at frequency f'. ■

Therefore, as an optimization, the initial estimate of the bounds, f_{init}^{low} and f_{init}^{high}, containing the minimum frequency can be set to $f_{init}^{low} = f_{init}^{high} = U_{cpu} * f_{max}$.

Based on the final frequency range $[f_{low}, f_{high}]$ returned by RatchetSearch, a binary search over the estimated range converges to the lowest frequency which allows a taskset to be schedulable, for a given schedulability-analysis technique.

Theorem 2: CycleSolo-CPU converges to the lowest CPU frequency f_{min} at which Γ is schedulable using analysis S.

Proof: The proof follows from Lemmas 1, 2, 3 and 4. ■

The time-complexity of RatchetSearch is linear in the number of tasks. The CycleSolo slack-calculation step has pseudo-polynomial complexity due to the response-time test.

Example: Consider a taskset Γ with two implicit-deadline tasks $\tau_1 = (C_1$=10, G_1=8, T_1=50), and $\tau_2 = (C_2$=20, G_2=5, T_2=80). For this example, assume that the CPU intervention required for accelerator access $G_m = 0$. To determine schedulability, we consider the request-driven analysis. The initial range estimates, f_{init}^{low} and f_{init}^{high}, are set to the total CPU utilization 0.45. For τ_1, the only effective scheduling point to consider is $t = 50$, which is τ_1's deadline. By calculating the CPU execution and slack up to time $t = 50$, we obtain the minimum CPU frequency estimate $f_1^{est} = C_1/(50 - G_1 - G_2) = 0.27$. However, as $f_1^{est} < f_{init}^{low}$ (Case I of RatchetSearch) the bounds are not updated. Subsequently,

we need to determine the lowest frequency f_2^{est}, which ensures τ_2 is schedulable in the presence of interference from τ_1, running at CPU frequency $f_1^{high} = 0.45$ (Lemma 3). From Lemma 2, the effective scheduling points we need to consider are $t = 42$, corresponding to an *effective* scheduling point of task τ_1, and $t = 80$, corresponding to τ_2's deadline. However, the request-driven analysis indicates that there is no slack up to $t' = 53$. Hence, we only need to consider $t = 80$. Computing the CPU execution and slack up to $t = 80$ yields $f_2^{est} = (2 * C_1 + C_2)/(80 - G_1 - G_2) = 0.597$. As $f_2^{est} > f_1^{high}$, the upper bound is updated to $f_2^{high} = 0.597$ (Case III of RatchetSearch), and the lower bound f_2^{low} is set to $f_1^{high} = 0.45$. Therefore, the final CPU frequency range is $[f_2^{low} = 0.45, f_2^{high} = 0.597]$. Performing a binary search over this range yields the minimum CPU frequency $f_{min} = 0.597$, ensuring schedulability using the request-driven analysis.

CycleSolo-Accel: Consider a uniprocessor whose frequency is not adjustable, with a single accelerator which supports frequency scaling. To minimize energy, we need to find the lowest-possible accelerator frequency at which all tasks meet their deadlines. In practice, such a scenario rarely exists, but CycleSolo-Accel helps bootstrap our CycleTandem algorithm.

Algorithm 10 presents the pseudo-code for CycleSolo-Accel, which considers tasks in decreasing priority order. For each task $\tau_i \in \Gamma$, CycleSolo-Accel uses RatchetSearch to compute the range $[f_i^{low}, f_i^{high}]$, which contains the lowest accelerator frequency which ensures that tasks $\tau_h \in \Gamma \mid h \leq i$ are schedulable. CycleSolo-Accel is identical to CycleSolo-CPU, except that it (i) calculates the frequency using the accelerator workload in the critical zone, and (ii) only performs the frequency-range estimation for tasks with accelerator segments. Thus, the results proved for CycleSolo-CPU also hold in the context of CycleSolo-Accel.

CycleSolo-ID: Consider a uniprocessor coupled with a single non-preemptive accelerator, where their operating frequencies can only be scaled by the same factor. To minimize energy, we need to find the lowest-possible *identical* frequency-scaling factor at which all tasks meet their deadlines. In practice, this scenario may exist when a CPU

Algorithm 10 Minimizing Accelerator Frequency

procedure CycleSolo-Accel(Γ, ϵ_{conv})
  $f_{low} = f_{high} = U_{acc}$ ▷ initial bounds
  for $\tau_i \in \Gamma$ **do** ▷ from high to low priority
    f_{low}, f_{high} = EstimateFreqRange($\tau_i, \Gamma, f_{high}, f_{low}$)
  f_{min} = BinarySearch($\Gamma, \epsilon_{conv}, f_{high}, f_{low}$)
  return f_{min}
procedure EstimateFreqRange($\tau_i, \Gamma, f_{high}, f_{low}$)
  /*S = slack, ω = resp time, β = accelerator workload*/
  $f_h = f_{high}$, $S = \beta = \Delta = 0$, $f_{est} = 1$, BusyFlag=TRUE
  W = Calculate-HP-ResponseTime(f_h)
  B_i = CalculateBlocking(τ_i, Γ, f_h, W)
  $\omega_g = G_i - G_{cpu}$ ▷ Accelerator execution time
  $\omega = C_i + G_i + B_i$, $\omega' = 0$, $J_i = \omega$
  while $\omega < D_i$ **do**
    if BusyFlag == TRUE **then**
      $\Delta = D_i - \omega$
      while $\omega < D_i$ AND $\Delta > 0$ **do**
        $\omega' = \sum_{h=0}^{i-1} E_h * [\lfloor \frac{\omega + W_h - (E_h)}{T_h} \rfloor + 1]$
        $\omega' = \omega' + J_i + S$, $\Delta = \omega' - \omega, \omega = \omega'$
      BusyFlag = FALSE
    else ▷ Start of an idle period
      t=Find-EarliestSchedulingPoint(τ_i, Γ, ω)
      $S = S + (t - \omega), \omega = t, t' = \omega_g + S, \beta = \omega_g$
      if $\beta / t' < f_{est}$ **then**
        $f_{est} = \beta / t'$
      BusyFlag = TRUE
  f_{low}, f_{high} = RatchetSearch-Step($f_{high}, f_{low}, f_{est}$)
  return f_{low}, f_{high}
procedure CalculateBlocking(τ_i, Γ, f_h, W)
  $G_{l,max} = max_{\tau_l \in lp(\tau_i)}(G_l)$, $B = G_{l,max} / f_h$, $B' = 0$
  while B != B' **do**
    $B' = B$, $B = \frac{G_{l,max}}{f_h} + \sum_{\tau_h \in hp(\tau_i)} \lceil \frac{B' + W_h - E_h}{T_h} \rceil * \frac{G_h}{f_h}$
  return B

is combined with an on-chip accelerator, and both share the same oscillator.

Algorithm 11 presents the pseudo-code for CycleSolo-ID, which considers tasks in decreasing order of their priority. For each task $\tau_i \in \Gamma$, CycleSolo-ID uses RatchetSearch to compute the range $[f_i^{low}, f_i^{high}]$, which contains the lowest common frequency-scaling

Algorithm 11 Minimizing the Common Frequency

procedure CycleSolo-ID(Γ, ϵ_{conv})
  $f_{low} = f_{high} = U_{cpu}$ ▷ initial bounds
  for $\tau_i \in \Gamma$ **do** ▷ from high to low priority
    f_{low}, f_{high} = EstimateFreqRange($\tau_i, \Gamma, f_{high}, f_{low}$)
  f_{min} = BinarySearch($\Gamma, \epsilon_{conv}, f_{high}, f_{low}$)
  return f_{min}
procedure EstimateFreqRange($\tau_i, \Gamma, f_{high}, f_{low}$)
  /*S = slack, ω = resp time, β = total workload*/
  $f_h = f_{high}$, $S = \beta = \Delta = 0$, $f_{est} = 1$, BusyFlag=TRUE
  W = Calculate-HP-ResponseTime(f_h)
  B_i = CalculateBlocking(τ_i, Γ, f_h, W)
  $\omega = C_i + G_i + B_i$, $\omega' = 0$, $J_i = \omega$
  while $\omega < D_i$ **do**
    if BusyFlag == TRUE **then**
      $\Delta = D_i - \omega$
      while $\omega < D_i$ AND $\Delta > 0$ **do**
        $\omega' = \sum_{h=0}^{i-1} E_h * [\lfloor \frac{\omega + W_h - (E_h/f_h)}{T_h} \rfloor + 1]$
        $\omega' = \omega' + J_i + S$, $\Delta = \omega' - \omega, \omega = \omega'$
      BusyFlag = FALSE
    else ▷ Start of an idle period
      t=Find-EarliestSchedulingPoint(τ_i, Γ, ω)
      $S = S + (t - \omega), \omega = t, t' = \omega - B_i, \beta = \omega - B_i - S$
      if $\beta/t' < f_{est}$ **then**
        $f_{est} = \beta/t'$
      BusyFlag = TRUE
  f_{low}, f_{high} = RatchetSearch-Step($f_{high}, f_{low}, f_{est}$)
  return f_{low}, f_{high}
procedure CalculateBlocking(τ_i, Γ, f_h, W)
  $G_{l,max} = max_{\tau_l \in lp(\tau_i)}(G_l)$, $B = G_{l,max}/f_h$, $B' = 0$
  while B != B' **do**
    $B' = B$, $B = \frac{G_{l,max}}{f_h} + \sum_{\tau_h \in hp(\tau_i)} \lceil \frac{B' + W_h - \frac{E_h}{f_h}}{T_h} \rceil * \frac{G_h}{f_h}$
  return B

factor which can ensure that tasks $\tau_h \in \Gamma \mid h \leq i$ are schedulable. CycleSolo-ID is identical to CycleSolo-CPU, except that it calculates the frequency using both the CPU and accelerator workload in the critical zone. Thus, all the results proved for CycleSolo-CPU also hold in the context of CycleSolo-ID.

Like SysClock, in theory, the CycleSolo algorithms can also use a per-task binary search

to converge to the minimum frequency f_i, which ensures that a task τ_i is schedulable in the presence of high-priority interference. This approach would entail multiple calls to the pseudo-polynomial response-time-analysis. Therefore, for n tasks and an operating frequency range f_{op}, the per-task-binary-search has $n * log(f_{op})$ pseudo-polynomial complexity. However, the ratchet-search-based-technique has $n + log(f_{rs})$ pseudo-polynomial complexity, where $f_{rs} < f_{op}$ is the range returned by RatchetSearch.

8.3 CycleTandem Algorithm

Consider a uniprocessor coupled with a single non-preemptive accelerator, where the CPU and accelerator operating frequencies can be set independently. Therefore, to minimize the energy consumption of the system, the CPU and accelerator frequencies need to be optimized in *tandem*.

Though the CPU and accelerator frequencies can be set independently, the taskset-schedulability constraint introduces a dependency between the CPU and accelerator frequencies. We now prove the "*See-Saw* Theorem" which shows the relationship between the CPU and accelerator frequencies.

Theorem 3: For a taskset Γ to be schedulable according to analysis S, both the CPU frequency f_{cpu} and the accelerator frequency f_{acc} cannot be less than the minimum common frequency, f_{id}^{solo}, determined by CycleSolo-ID.

Proof: Suppose Γ is schedulable by S, when both the CPU and accelerator frequencies $f_{cpu}, f_{acc} < f_{id}^{solo}$. Therefore, Γ will be schedulable if both the accelerator and CPU frequencies are set to $f_{id} = max(f_{cpu}, f_{acc}) < f_{id}^{solo}$. However, given an analysis technique S, CycleSolo-ID returns the minimum common frequency, f_{id}^{solo}. Thus, there is a contradiction. ■

We now see that there is a *see-saw* relationship between f_{acc} and f_{cpu}. If one is reduced to be less than f_{id}^{solo}, the other will increase and *always* be greater than or equal to f_{id}^{solo}. Thus, we can also conclude that f_{acc} is monotonically non-increasing with f_{cpu}, and vice-

Algorithm 12 CycleTandem Algorithm

```
1: procedure CycleTandem(Γ, ε_conv)
2:     f_cpu^solo = CycleSolo-CPU(Γ, ε_conv)
3:     f_acc^solo = CycleSolo-Accel(Γ, ε_conv)
4:     f_cpu^up, f_acc^up = ComputeRange(Γ, f_cpu^solo, f_acc^solo, ε_conv)
5:     if f_cpu^up − f_cpu^solo < f_acc^up − f_acc^solo then
6:         f_cpu, f_acc = SearchRange-CPU(Γ, f_cpu^up, f_cpu^solo)
7:     else
8:         f_cpu, f_acc = SearchRange-Acc(Γ, f_acc^up, f_acc^solo)
9:     return f_cpu, f_acc
```

versa. Therefore, given an analysis technique, for every *feasible* CPU frequency f_{cpu}, there exists a *unique* minimum accelerator frequency, f_{acc}, which guarantees schedulability. This accelerator frequency can be found by scaling the taskset to the CPU frequency, and then computing the corresponding minimum accelerator frequency using CycleSolo-Accel. Similarly, for every *feasible* accelerator frequency f_{acc}, CycleSolo-CPU can find the minimum CPU frequency, f_{cpu}, guaranteeing schedulability.

It is trivial to show that, for a fixed CPU (accelerator) frequency, the minimum corresponding accelerator (CPU) frequency minimizes energy. Therefore, given the one-to-one *see-saw* relationship between the CPU and accelerator frequencies, it is sufficient to find the *optimal* accelerator frequency in order to compute the *optimal* CPU frequency, and vice versa. Thus, an exhaustive search in the range of feasible accelerator frequencies will yield the optimal frequency pair $(f_{cpu}^{opt}, f_{acc}^{opt})$.

Theorem 4: The energy-optimal accelerator frequency f_{acc}^{opt} always lies in $[f_{acc}^{solo}, f_{acc}^{up}]$, where f_{acc}^{solo} is the frequency returned by CycleSolo-Accel, and f_{acc}^{up} is the minimum feasible accelerator frequency guaranteeing schedulability, when the CPU is set to the CycleSolo-CPU frequency f_{cpu}^{solo}.

Proof: If we set the CPU frequency to be f_{cpu}^{solo}, there may be some slack in the system which can be utilized to reduce the accelerator frequency below its maxima. As f_{cpu}^{solo} is the lowest feasible CPU frequency (Theorem 2), the corresponding accelerator frequency f_{acc}^{up} can safely upper-bound the range which contains the optimal accelerator frequency,

as any frequency above f_{acc}^{up} will not be energy-efficient. Additionally, CycleSolo-Accel yields the lowest feasible accelerator frequency f_{acc}^{solo} (corollary of Theorem 2). Therefore, the optimal accelerator frequency f_{acc}^{opt} must lie in $[f_{acc}^{solo}, f_{acc}^{up}]$. ■

As a corollary, if we choose the accelerator frequency to be f_{acc}^{solo}, we obtain the CPU frequency f_{cpu}^{up}. Therefore, the optimal CPU frequency f_{cpu}^{opt} will lie in $[f_{cpu}^{solo}, f_{cpu}^{up}]$.

For CycleSolo, assuming a convex energy function, the energy consumption is minimized at the minimum *feasible* frequency, which is independent of the power-model parameters. However, for CycleTandem, the *energy-optimal* frequency pair depends on the power-model parameters, due to the non-linear see-saw schedulability-analysis-dependent relationship between the accelerator and CPU frequencies which renders the energy function non-convex. This non-linearity is caused by an effect we refer to as *slack-squeezing*. Given some usable slack, the frequency of the CPU (accelerator) depends on the effective utilization of the CPU (accelerator) up to a scheduling point. Therefore, a $\delta > 0$ increase in the CPU (accelerator) frequency can cause a $\delta' > \delta$ decrease in the corresponding accelerator (CPU) frequency. In other words, if $\delta_{cpu} > \delta_{acc}$, the CPU *squeezes* the available slack more efficiently than the accelerator, and vice versa. The following theorem highlights the effect of slack squeezing on energy.

Theorem 5: Consider a feasible CPU frequency $f_{cpu} \in [f_{cpu}^{solo}, f_{max})$, and its corresponding minimum feasible accelerator frequency f_{acc}. If we increase the CPU frequency by δ, then the energy consumption decreases *if and only if* the gradient of the accelerator frequency with respect to the CPU frequency,

$$|\nabla_{f_{cpu},f_{acc}}| = |\frac{\partial f_{acc}}{\partial f_{cpu}}| > \frac{K_{cpu} * U_{cpu} * f_{cpu}^{\alpha-1}}{K_{acc} * U_{acc} * f_{acc}^{\alpha-1}} \tag{8.5}$$

Proof: Given the power model from Section 8.1.2, the gradients of the energy at any frequency pair (f_{cpu}, f_{acc}), are: $\partial E_{total}/\partial f_{cpu} = \alpha * K_{cpu} * U_{cpu} * f_{cpu}^{\alpha-1}$ and $\partial E_{total}/\partial f_{acc} = \alpha * K_{acc} * U_{acc} * f_{acc}^{\alpha-1}$. Now, for each frequency pair, assume that an increase in one frequency say f_{cpu}, causes a corresponding increase in energy (∂E_{total}), which can be compensated ($-\partial E_{total}$) by a decrease in the other frequency f_{acc}. In this case, we can

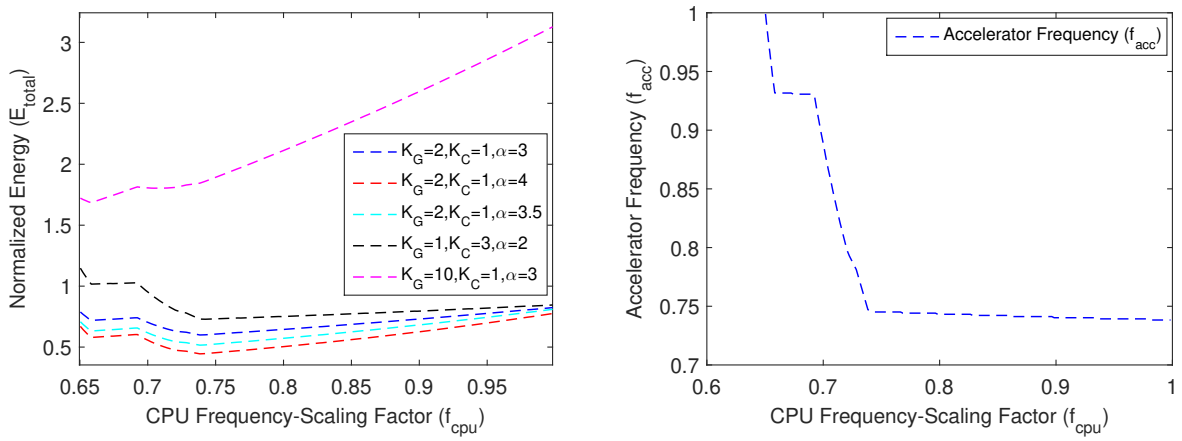
divide one of the above gradients by the other, we get the ratio of the frequency changes which yields the same energy, i.e.,

$$|\nabla_0| = \frac{\partial E_{total}/\partial f_{cpu}}{\partial E_{total}/\partial f_{acc}} = \frac{\partial f_{acc}}{\partial f_{cpu}} = \frac{K_{cpu} * U_{cpu} * f_{cpu}^{\alpha-1}}{K_{acc} * U_{acc} * f_{acc}^{\alpha-1}}$$

Now, for a feasible frequency pair (f_{cpu}, f_{acc}), if the absolute value of the gradient of the *see-saw* relationship at frequency f_{cpu}, $|\nabla_{f_{cpu},f_{acc}}| > |\nabla_0|$, then the decrease in the accelerator frequency f_{acc} can more than compensate for the increase in energy caused by increasing the CPU frequency f_{cpu}, causing the overall energy to decrease. Conversely, if the energy decreases on increasing the CPU frequency f_{cpu} by a small value Δ, then the corresponding decrease in accelerator frequency f_{acc} by a small value Δ', is more than that required to compensate for the increase in energy caused by increasing f_{cpu}, leading to $|\nabla_{f_{cpu},f_{acc}}| > |\nabla_0|$. ■

The corollary of the above theorem, corresponding to the accelerator frequency also holds.

Consider a taskset Γ with CPU and accelerator utilization U_{cpu} and U_{acc}. Let (f_{cpu}, f_{acc}) constitute a feasible frequency pair, ensuring Γ is schedulable. Therefore, based on the energy model described in Section 8.1.2, the *normalized* energy consumption of the system in the hyperperiod is given by $E_{total} = E_{cpu} + E_{acc}$, where $E_{cpu} = K_{cpu} * U_{cpu} * f_{cpu}^{\alpha-1} + E_{static}^{cpu}$ is the CPU energy, and $E_{acc} = K_{acc} * U_{acc} * f_{acc}^{\alpha-1} + E_{static}^{acc}$ is the accelerator energy. Since we have no control on reducing the static power of the system, we conservatively assume that it also shares a direct dependency on the operating voltage and frequency. Thus, both E_{cpu} and E_{acc} are indeed convex non-decreasing functions in f_{cpu} and f_{acc} respectively, and their combination E_{total} will also be a convex function in f_{cpu} and f_{acc}. However, if the schedulability constraint is applied, then due to the *see-saw* theorem, f_{acc} can become a non-linear non-convex function of f_{cpu} and vice-versa, which in turn can render the energy function E_{total} non-convex. This non-convexity unfortunately makes it difficult to find the optimal frequency-pair which minimizes the total energy consumption.



(a) (b)

Figure 8.1: (a) Energy and (b) Accelerator Frequency vs CPU Frequency

By looking at the function for $E_{total} \forall (\alpha > 3)$, it may seem that, if $K_{cpu} * U_{cpu} > K_{acc} * U_{acc}$, then to minimize E_{total}, $f_{cpu} < f_{acc}$, and vice versa. However, the non-linear relationship between f_{acc} and f_{cpu} causes this statement to not always hold, except under special conditions.

Theorem 6: If $K_{cpu} * U_{cpu} >> K_{acc} * U_{acc}$, then the energy-optimal CPU frequency for taskset Γ approaches f_{cpu}^{solo}.

Proof: In this case, $E_{total} \approx E_{cpu}$. As E_{cpu} is minimized at f_{cpu}^{solo}, therefore as $f_{cpu} \rightarrow f_{cpu}^{solo}$, E_{total} is also minimized. ■

As a corollary, if $K_{acc} * U_{acc} >> K_{cpu} * U_{cpu}$, then for taskset Γ the energy-optimal accelerator frequency $f_{acc} \rightarrow f_{acc}^{solo}$. Therefore, the energy-optimal frequency pair depends on (i) the ratio of the accelerator and CPU utilization, (ii) the power-model parameters, and (iii) the schedulability-analysis-dependent see-saw relationship between f_{acc} and f_{cpu}.

Example: Consider a taskset Γ consisting of 5 implicit-deadline tasks with their execution parameters (C, G_e, G_m, T): $\{(0.27, 15.11, 1, 84), (24.08, 0, 0, 221), (37.89, 0, 0, 231), (13.19, 0, 0, 330), (78.44, 45.18, 1, 427)\}$. Figure 8.1 illustrates the relationship of the CPU frequency, f_{cpu} with the energy consumption Figure 8.1a and the accelerator frequency (Figure 8.1b). In (Figure 8.1a), note the non-convexity of the energy function and the

dependence of the energy-optimal frequency on the power-model parameters. Additionally, observe the non-linear relationship between the accelerator frequency and the CPU frequency in Figure 8.1b. Notice that for $f_{cpu} \approx 0.7$ to 0.75, the accelerator frequency drops rapidly due to the slack-squeezing effect.

It appears that the energy-optimal frequency pair cannot be found analytically, and instead we need to search the *feasible* space to find the optimal solution. Of the CPU and accelerator frequency ranges, the smaller range can be chosen. In theory, a gradient descent over the feasible range can yield a local optima. Alternatively, if the range is small, an exhaustive search with a small step size can quickly yield a good solution.

Algorithm 12 presents the CycleTandem algorithm, which can utilize any reasonable search technique, SearchRange, in the feasible range. In our experiments, we use a greedy-search algorithm which first computes the energy at both ends of the feasible range. Subsequently, it chooses the endpoint with the lower energy, and increases/decreases the CPU (accelerator) frequency in small steps, until the first local minima is reached.

Accommodating Discrete Frequencies: In most systems, the frequency of the processor (accelerator) can only be set to discrete values. For CycleSolo, we can compute the minimum frequency, and pick the next-greater discrete frequency. Alternatively, the binary-search step of CycleSolo can only consider the discrete frequencies. If no discrete frequencies lie in the feasible range, we pick the next-greater frequency than the upper bound of the range. Similarly, for CycleTandem, we can search over the discrete frequencies in the feasible range, and pick the frequency pair yielding the minimum energy.

8.4 Multi-core CycleSolo and CycleTandem

We now extend CycleSolo and CycleTandem for fully-partitioned multi-core processors, coupled with a single accelerator. With respect to CPU frequency, we consider two scenarios where, if the CPU supports frequency scaling, then (i) all its cores *must* be set

to the same frequency, and (ii) each core's frequency can be independently set. We call case (i) the *uniform-frequency* setting, and case (ii) the *independent-frequency* setting. We now propose techniques for both of these scenarios. Lastly, we discuss task-partitioning techniques for both scenarios.

8.4.1 Uniform Frequency

Consider a taskset Γ, partitioned among m cores, such that each core has a subset of tasks $\Psi_j \subset \Gamma \mid j = 1, 2, .., m$. The context in which each algorithm can be used, given the uniform-frequency setting is as follows:

1) CycleSolo-CPU: all the CPU cores can be set to a common frequency, and the accelerator frequency is not adjustable.

2) CycleSolo-Accel: the frequency of the CPU cores cannot be scaled, and the accelerator frequency is adjustable.

3) CycleSolo-ID: all the CPU cores and the accelerator, can only be set to an identical frequency-scaling factor.

4) CycleTandem: all the CPU cores can be set to a common frequency, and the accelerator frequency is also adjustable.

Consider the CycleSolo algorithms. The best solution is still a *single* lowest frequency. Therefore, Lemma 3 holds, and we can use the RatchetSearch frequency-range estimation algorithm. The only differences from the uniprocessor case are as follows: (i) for each core j, interference is computed considering only the tasks $\tau_i \in \Psi_j$, and (ii) remote blocking by tasks on other cores is taken into account.

Theorem 7: Given a taskset Γ, and its schedulable partition Ψ onto an m-core processor, then the CycleSolo algorithms converge to the lowest CPU/accelerator/common frequency f_{min} at which partition Ψ is schedulable using analysis S.

Proof: The CycleSolo algorithms compute a single frequency range. Thus, Theorem 2 holds, and a *single* binary search, testing schedulability across m cores using analysis

S, over the feasible range returned by RatchetSearch, converges to the lowest frequency f_{min}. ■

Consider the CycleTandem algorithm. We share a single CPU frequency across all cores. Therefore, the *see-saw* theorem still holds, and a one to one mapping between every common CPU and accelerator frequency still exists. Thus, multi-core CycleTandem can use CycleSolo-CPU and CycleSolo-Accel to compute a lower bound on the common CPU frequency f_{cpu}^{solo}, and the accelerator frequency f_{acc}^{solo} respectively. Therefore, using f_{cpu}^{solo} and f_{acc}^{solo}, the safe upper bounds on the accelerator frequency f_{acc}^{up}, and the common CPU frequency f_{cpu}^{up} can be calculated. Subsequently, performing an exhaustive search over the feasible CPU or accelerator range can yield the *energy-optimal* frequency pair. Similarly, we can also use the greedy search proposed for uniprocessors.

8.4.2 Per-CPU-core Independent Frequency

Consider a taskset Γ, partitioned among m cores, such that each core has a subset of tasks $\Psi_j \subset \Gamma \mid j = 1, 2, .., m$. In the independent-frequency setting, where each CPU core can have its own frequency, only the CycleSolo-CPU and CycleTandem algorithms can exploit this feature. Therefore, we now describe techniques for both the CycleSolo-CPU and CycleTandem scenarios in the independent-frequency setting.

Due to self-suspension, the response time of a task on one core, can depend on the frequency chosen for another core. This is due to the dependence of a task τ_i's response time on the response time of higher-priority tasks $\tau_h \mid h < i$, which may be allocated to other cores. This interdependence between cores makes it difficult to find the optimal frequencies for a given task partition in the independent-frequency setting. Therefore, we cannot perform a search over a *single* frequency range, and instead need to rely on heuristics to find a good solution.

Algorithm 13 CycleSolo-CPU Independent Frequency

procedure CycleSolo-CPU-Independent($\Gamma, \Psi, m, \epsilon_{conv}$)
  /* Ψ - Partition, m - Number of Cores*/
  $\chi = \{\}$ ▷ set of cores whose frequency is chosen
  f_{cpu}^{solo} = CycleSolo-CPU(Γ, ϵ_{conv}) ▷ CycleSolo-CPU Uniform Frequency
  $f_{cpu,j} = f_{cpu}^{solo} \forall j \in 1, 2, .., m$ ▷ set each core's frequency to f_{cpu}^{solo}
  for $j \in 1$ to m **do** ▷ choose the best core in each iteration
    $f, core$ = FindBestCoreFreq(Γ, χ, f_{cpu})
    $f_{cpu,j} = f$ ▷ set the chosen frequency for the chosen core
    $\chi \leftarrow \chi$ + core ▷ assign the core to chosen set
  return $f_{cpu,j} \forall j \in 1, 2, .., m$
procedure FindBestCoreFreq(Γ, χ, f_{cpu})
  /* χ - cores whose frequency is chosen */
  /* f_{cpu} - array of assigned per-core frequencies */
  /* Initialize the best energy savings, core and frequency */
  $E_{saved}^{best} = 0$, $core_{best} = -1$, $f_{best} = f_{cpu}^{solo}$
  for $j \in 1$ to m **do** ▷ find core with most energy-reduction capability
    if $j \notin \chi$ **then** ▷ check if core frequency is not set yet
      $f_{high} = f_{cpu}^{solo}$, $f_{low} = U_{cpu,j}$ ▷ set search range
      /* scale the tasks as per the assigned frequencies */
      Γ' = ScaleTasksetCPU-Frequency(f_{cpu})
      /* estimate the best frequency for this core */
      f_{chosen} = BinarySearch($\Gamma', \epsilon_{conv}, f_{high}, f_{low}$)
      E_{saved} = EstimateEnergySaving(Γ, f_{chosen})
      /* check if the energy savings is the best in this iteration */
      if $E_{saved} > E_{saved}^{best}$ **then**
        $E_{saved}^{best} = E_{saved}$, $core_{best} = j$, $f_{best} = f_{chosen}$
  return $f_{best}, core_{best}$

CycleSolo-CPU Independent Frequency

In this scenario, each CPU core can be individually set to an independent frequency, and the accelerator frequency is not adjustable. To find a good set of per-core frequencies, $f_{cpu,j}^{solo} | j < m$, we utilize a greedy heuristic described in Algorithm 13. This heuristic initially starts by setting each core's frequency to the CycleSolo-CPU uniform frequency, $f_{cpu,j} = f_{cpu}^{solo} | j < m$. We then try to estimate the lowest frequency $f_{cpu,j}$ each core j can be set to, while keeping all the other cores at a fixed frequency. The core k which leads to the most energy reduction is chosen and added to the set χ, which keeps track of

Algorithm 14 CycleTandem-Independent Algorithm

1: **procedure** CycleTandem-Independent($\Gamma, \Psi, m, \epsilon_{conv}$)
2: f_{cpu}^{solo} = CycleSolo-CPU(Γ, ϵ_{conv}) ▷ get the CycleSolo-CPU uniform frequency
3: f_{acc}^{solo} = CycleSolo-Accel(Γ, ϵ_{conv}) ▷ get the CycleSolo-Accel frequency
4: $f_{cpu}^{up}, f_{acc}^{up}$ = ComputeRange($\Gamma, f_{cpu}^{solo}, f_{acc}^{solo}, \epsilon_{conv}$) ▷ compute the search range
5: /* Perform a search over the Accelerator Frequency Range */
6: $\{f_{cpu,j} \forall j \in 1, 2, .., m\}, f_{acc}$ = SearchRangeIndependent-Acc($\Gamma, f_{acc}^{up}, f_{acc}^{solo}$)
7: **return** $\{f_{cpu,j} \forall j \in 1, 2, .., m\}, f_{acc}$ ▷ per-core CPU and accelerator frequencies

the cores whose frequency has already been assigned, and the frequency of core $f_{cpu,k}$ is set to the corresponding frequency. This process is repeated m times, where m is the number of cores, such that in each iteration, (i) only cores $j \notin \chi$ are considered in the frequency-estimation phase with their initial frequency assigned as f_{cpu}^{solo}, and (ii) the cores $k \in \chi$ are set to the frequency that was assigned to them in previous iterations. This ultimately yields a solution where, each core j's frequency is $f_{cpu,j} <= f_{cpu}^{solo} | j < m$. Note that the solution provided by this heuristic is not optimal, as in some situations, increasing the frequency of some core k can provide greater energy savings due to the decrease in the frequency of other cores $j \neq k$. However, the heuristic guarantees that the energy consumption of the solution is lower than or equal to the uniform-frequency CycleSolo-CPU solution.

CycleTandem Independent Frequency

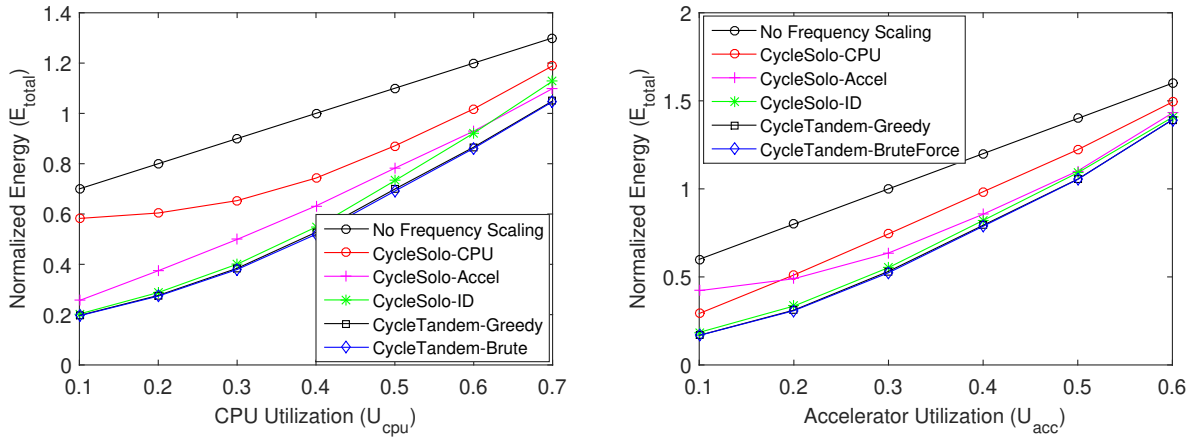
In this scenario, each CPU core can be individually set to an independent frequency, and the accelerator frequency is also adjustable. To find a good set of per-core frequencies, $f_{cpu,j}^{tandem} | j < m$, we utilize a heuristic described in Algorithm 14. Like the uniform-frequency CycleTandem setting, this heuristic also performs a greedy or brute-force search over the feasible accelerator frequency range. For a feasible accelerator frequency in the range, the taskset is scaled, and the CycleSolo-CPU-Independent heuristic (Algorithm 13) is used to find a set of good per-core CPU frequencies, corresponding to the chosen accelerator frequency.

8.4.3 Task Partitioning

Load balancing is often used to determine an energy-efficient partition in multi-core systems [79]. When the CPU frequency is common across all cores, the core with the highest *effective* load determines the CPU frequency. Thus, load balancing is useful as it tries to minimize the maximum load across cores [79]. Among task-partitioning heuristics studied in the literature, the *Worst-Fit Decreasing* (WFD) heuristic is known to yield a well-balanced partition [79]. WFD allocates tasks to cores in non-decreasing order of their utilization. Given a task to be allocated, WFD assigns it to the core with the least utilization. When WFD can allocate tasks to use only m cores, it is equivalent to *List Scheduling*.

However, the blocking and self-suspension penalties introduced by accessing the accelerator affects the frequency estimation. Therefore, motivated by the work in [182], we propose a modified version of WFD called *Sync-Aware* WFD or SA-WFD, which for a taskset Γ, first computes the fraction of CPU load belonging to tasks which utilize the accelerator, $\gamma_{acc} = U_{cpu}^{acc}/U_{cpu}^{total}$, and subsequently allocates $\psi = \lceil \gamma_{acc} * m \rceil$ cores for these tasks. Finally, our heuristic balances the load while constraining the tasks using the accelerator to ψ cores. Thus, this heuristic restricts the self-suspension penalties to a few cores.

Similarly, when each core can have its own frequency (CycleSolo-CPU and CycleTandem), load balancing ensures that all cores have an intermediate frequency, rather than a single core having a high frequency. This is desirable, as indicated in the power model in Section 8.1.2, energy consumption is proportional to f^{α}, where $\alpha > 3$ [83]. Therefore, both WFD and SA-WFD are also applicable in this setting.



(a) (b)

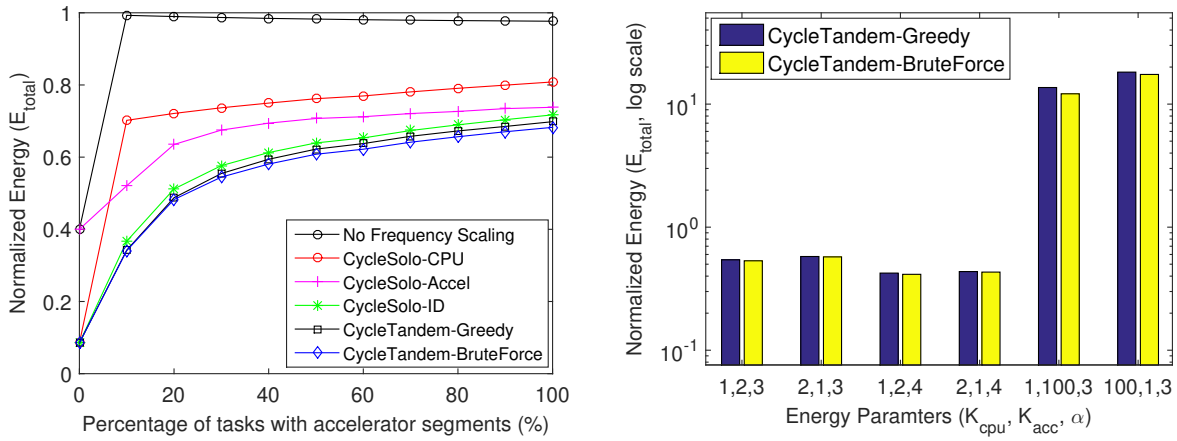
Figure 8.2: Energy as a function of (a) CPU Utilization, (b) Accelerator Utilization

8.5 Experimental Evaluation

We now assess the energy-savings delivered by our proposed CycleSolo and CycleTandem algorithms. We first present analytical evaluations using the request-driven analysis [55] and the power model presented in Section 8.1.2. Subsequently, we present experiments performed on the NVIDIA TX2 [22] embedded platform to demonstrate the practical applicability of our proposed techniques. We assume fully-partitioned fixed-priority scheduling, with task priorities assigned using the Rate-Monotonic policy [29]. To the best of our knowledge, no other energy-saving real-time scheduling techniques exist in the context of hardware accelerators. Therefore, we compare against a base case without energy management.

8.5.1 Analytical Evaluation

We compare our proposed techniques on the basis of analytically-computed energy savings over the hyperperiod of a given taskset. Every data point plotted is an average of 5000 tasksets, randomly generated using the UUniFast-Discard [175] algorithm, such that no task has a CPU/accelerator utilization greater than 0.4. We consider sporadic tasks with the minimum inter-arrival time randomly assigned to be between 5 and 500



(a) (b)

Figure 8.3: Energy as a function of (a) % of tasks using the accelerator (b) power-model parameters

time units.

Uniprocessor Experiments: Figures 8.2a, 8.2b and 8.3a plot the average normalized energy as we vary (a) the CPU utilization keeping $U_{acc} = 0.3$, (b) the accelerator utilization keeping $U_{cpu} = 0.4$, and (c) the fraction of tasks using the accelerator keeping $U_{cpu} = 0.4$ and $U_{acc} = 0.3$. The power-model parameters are set to: $K_{cpu} = 1$, $K_{acc} = 2$ and $\alpha = 3$. Consider Figure 8.2a. As the CPU utilization is varied, our proposed CycleTandem greedy-search heuristic returns a solution that in the worst-case consumes 1.53% greater energy than the brute-force search. Compared to the case without energy management, i.e., executing all tasks at the maximum frequency, CycleTandem with the greedy-search heuristic on average delivers up to 71.88% lower energy consumption. Compared to the case without energy management, CycleSolo-CPU, CycleSolo-Accel and CycleSolo-ID deliver up to 27.42%, 63.41% and 71.03% lower energy respectively. Similar trends can be observed for Figures 8.2b and 8.3a, where the CycleTandem greedy-search heuristic yields a result with near-optimal energy savings.

Figure 8.4 shows the computed CPU and accelerator frequencies for our proposed techniques. For all three sub-figures, (8.4a, 8.4b and 8.4c) the CycleSolo-Accel and CycleSolo-CPU frequencies are never greater than the CycleSolo-ID frequency. For Cy-

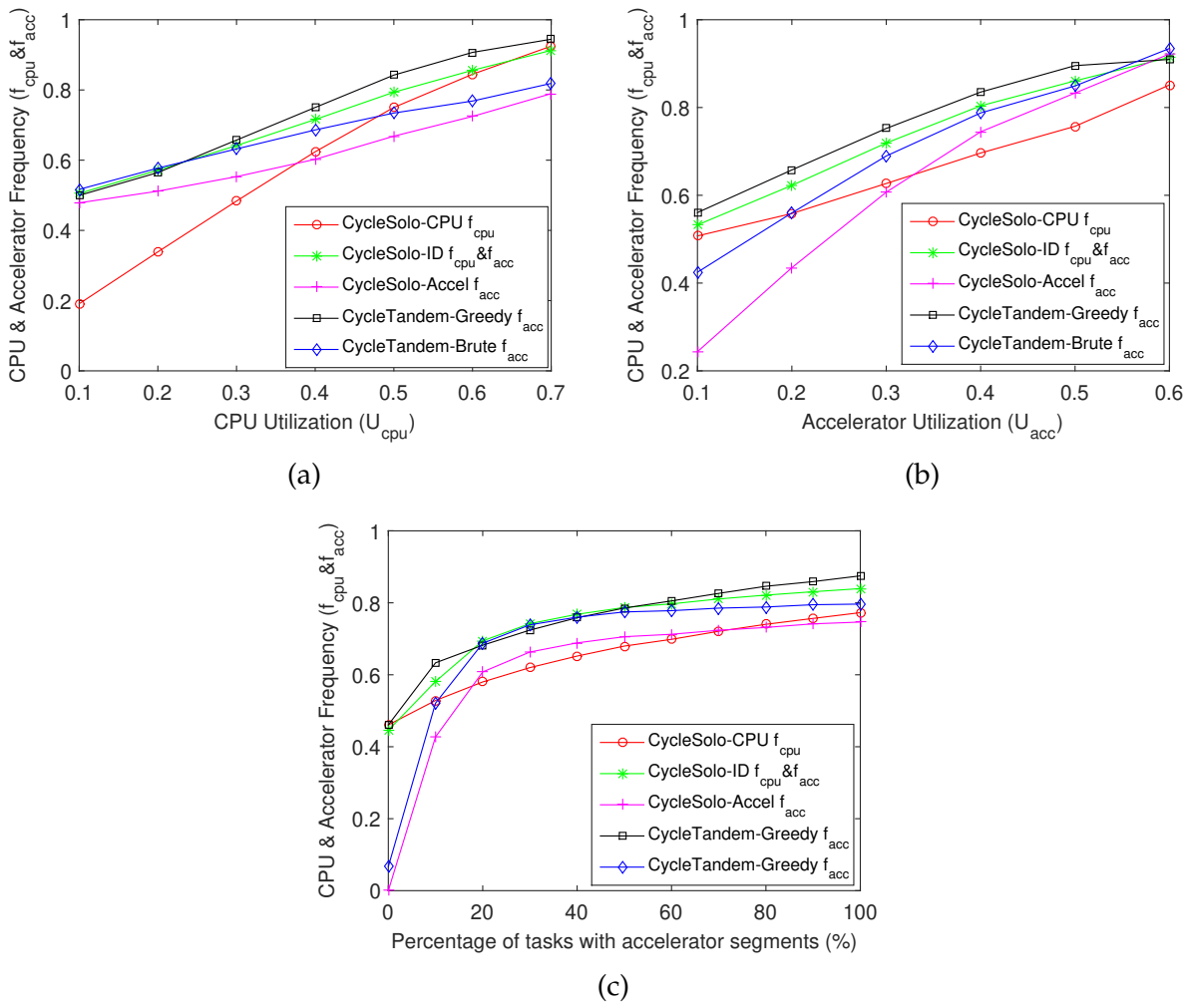


Figure 8.4: CPU and Accelerator frequency as a function of (a) CPU Utilization, (b) Accelerator Utilization, and (c) Percentage of tasks using the accelerator

cleTandem, note the see-saw relationship between the accelerator and CPU frequencies around the CycleSolo-ID frequency.

We also performed experiments to determine the impact of varying the power-model parameters on the CycleTandem greedy-search heuristic. Figure 8.3b plots the average normalized energy for five sets of power-model parameters. Note that when the power-model parameters are varied, the greedy-search heuristic yields solutions that, in the worst case, consume only up to 12.48% more energy than the brute-force search.

Multicore Experiments Uniform Frequency: In Section 8.4, we proved that given a taskset Γ and its schedulable partition P onto an m-core processor, we can find the

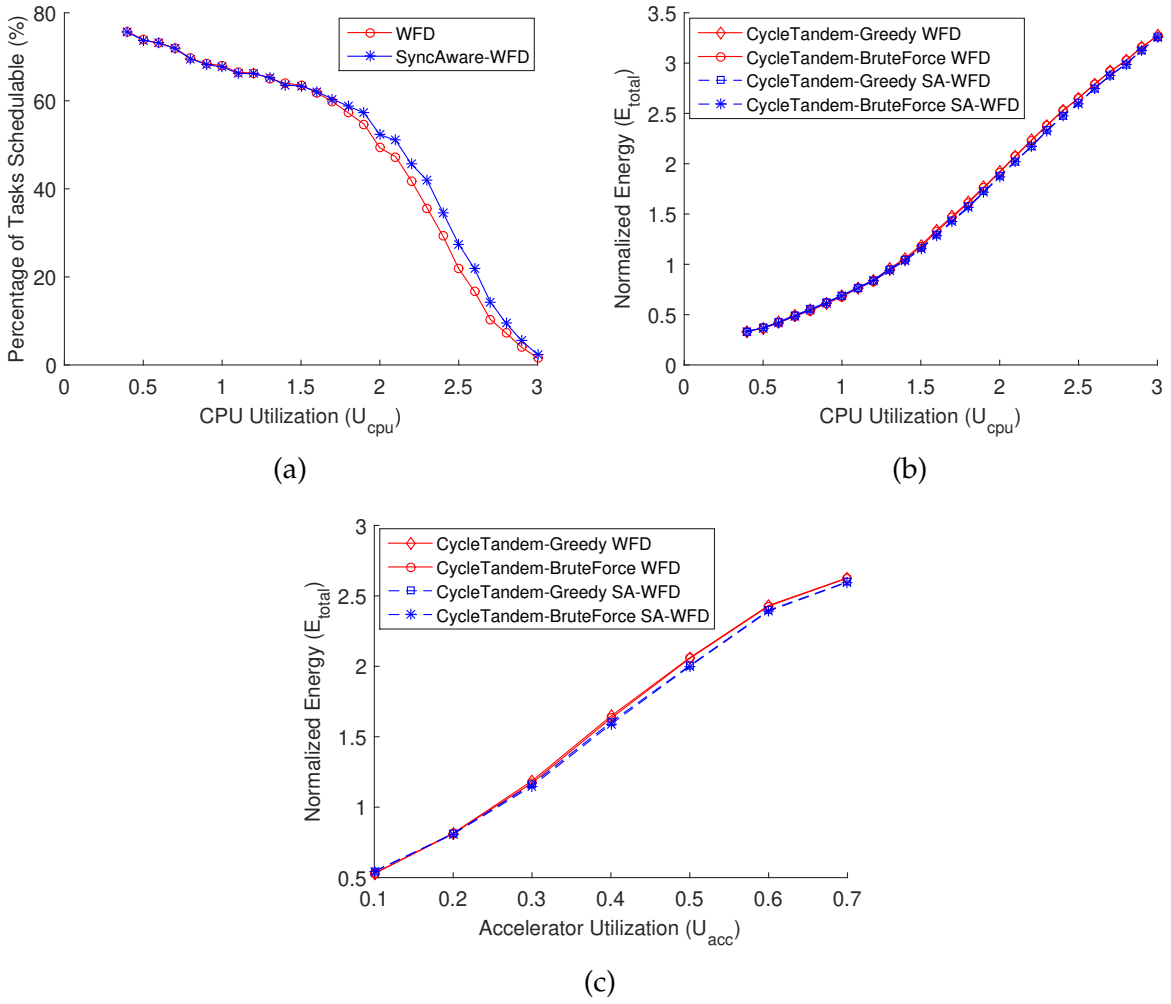


Figure 8.5: WFD vs Sync-Aware WFD (SA-WFD) (a) schedulability comparisons, (b) Energy vs CPU utilization, and (c) Energy vs Accelerator utilization

optimal solution for all the CycleSolo algorithms. Therefore, we consider $m = 4$ cores, and focus on comparing our proposed Sync-Aware WFD (SA-WFD) heuristic against WFD, in the context of CycleTandem.

Figure 8.5a compares SA-WFD and WFD on the basis of schedulability. The CPU utilization is varied while keeping $U_{acc} = 0.3$. Observe that, as the CPU utilization increases, SA-WFD yields more schedulable partitions than WFD, and can schedule up to 6.3% more tasksets than WFD. We also illustrate the utility of our proposed greedy-search heuristic for CycleTandem. Figures 8.5b and 8.5c plot the average normalized energy for CycleTandem as we vary (b) the CPU utilization keeping $U_{acc} = 0.3$, and (c)

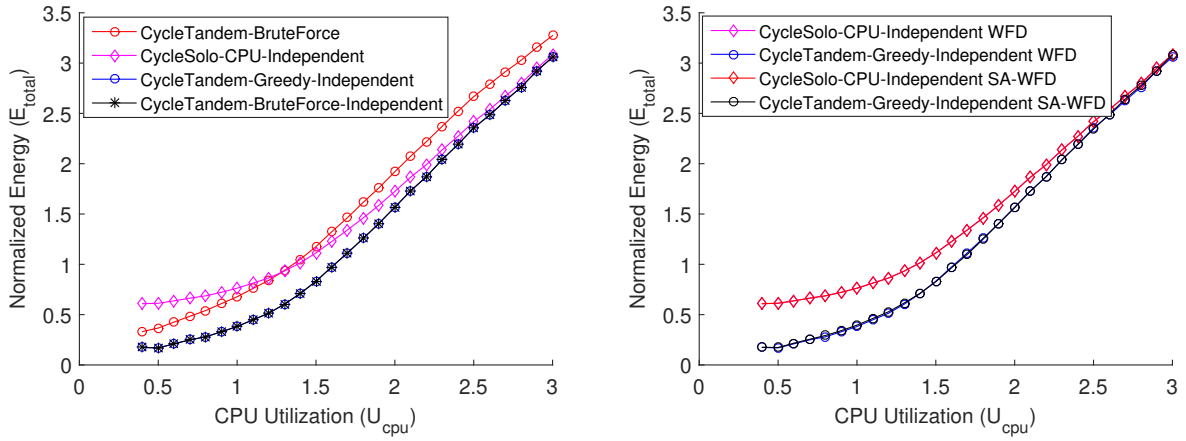


Figure 8.6: Multicore Independent Frequency normalized energy vs CPU utilization (a) independent frequency vs uniform frequency, and (b) WFD vs SA-WFD

the accelerator utilization keeping $U_{cpu} = 1.5$. Note that, for both WFD and SA-WFD, the CycleTandem greedy-search heuristic returns a solution that, in the worst case consumes 1.38% and 1.42% more energy than the brute-force search, respectively. Comparing the CycleTandem greedy-search heuristic across the two partitioning techniques indicates that SA-WFD on average yields a solution with up to 3.3% lower energy consumption than WFD. Thus, SA-WFD yields better schedulability and energy savings than WFD, with the same algorithmic complexity.

Multicore Experiments Independent Frequency: In Section 8.4.2, we stated that the independent-frequency setting is only applicable to CycleSolo-CPU and CycleTandem. Therefore, we consider $m = 4$ cores, and focus on (i) comparing the energy savings of these two algorithms in the independent-frequency setting as compared to the uniform-frequency setting, and (ii) comparing our proposed Sync-Aware WFD (SA-WFD) heuristic against WFD, for both CycleSolo-CPU and CycleTandem in the independent-frequency context.

Figures 8.6a and 8.6b plot the normalized energy as we vary the CPU utilization for a quad-core processor. For the uniform-frequency multicore setting, CycleTandem yields the most energy-efficient solution. Therefore, in Figure 8.6a, we compare the

Table 8.1: Experimental Tasksets Deployed on the NVIDIA TX2

Taskset (Γ)	Tasks	U_{cpu}	U_{acc}	Description
Γ_1	8	1.75	0.39	High U_{cpu}, High U_{acc}
Γ_2	4	0.44	0.14	Low U_{cpu}, High U_{acc}
Γ_3	4	1.94	0.02	High U_{cpu}, Low U_{acc}
Γ_4	6	0.10	0.52	Low U_{cpu}, High U_{acc}

independent-frequency heuristics using the uniform-frequency CycleTandem solution as the baseline, while using the WFD partitioning algorithm. We observe that, on average, if we can set independent frequencies on each core both CycleSolo-CPU-Independent and CycleTandem-Independent, can yield solutions with up to 10.3% and 53.8% lower energy consumption than the uniform-frequency CycleTandem solution respectively. In Figure 8.6b, we compare the independent frequency algorithms across the two partitioning techniques: WFD and SA-WFD. Our experiments indicate that SA-WFD can yield marginally lower energy consumption (on average only up to 1.33%).

8.5.2 Experiments on the NVIDIA TX2

We next examine the practical energy savings delivered by our algorithms by performing experiments on the NVIDIA Jetson TX2 embedded platform [22]. The TX2 contains 4 ARM A57 CPU cores, 2 Denver CPU cores and an integrated 256-core Pascal GPU [22]. In our experiments, we disable the 2 Denver CPU cores. The ARM cores can be set to 12 discrete frequencies ranging from 345.6 MHz to 2.03 GHz, and the GPU can be set to 12 discrete frequencies ranging from 114.75 MHz to 1.13 GHz. Note that all 4 ARM cores lie in the same voltage domain, and can *only* be set to the same frequency (uniform frequency). However, the GPU lies in a separate power domain and its frequency can be independently set.

We consider four tasksets $\Gamma_{i|i=1,2,3,4}$ described in Tables 8.1 and 8.2. Each taskset consists of matrix-multiplication tasks accessing the GPU. The length of each task's CPU and GPU segments are configurable. An implementation of the MPCP-based synchro-

Table 8.2: NVIDIA TX2 Taskset Parameters (ms)

Γ	Implicit-deadline $(D = T)$ Task Parameters (C,G_e,G_m,T)
Γ_1	(20,4,0.3,100),(10,6,0.3,100),(30,4,0.3,150),(50,18,0.3,200), (50,9,0.3,300),(100,30,0.3,300),(100,13,0.3,600),(400,30,0.3,1200)
Γ_2	(10,13,0.3,150),(50,4,0.3,300),(60,18,0.3,600),(125,9,0.3,1200)
Γ_3	(200,4,0.3,450),(300,4,0.3,600),(400,4,0.3,900),(1000,4,0.3,1800)
Γ_4	(2,18,0.3,250),(4,30,0.3,250),(10,47,0.3,500),(20,47,0.3,500), (10,89,0.3,750),(30,30,0.3,1500)

Table 8.3: CPU,GPU Frequencies used on the NVIDIA TX2 (MHz)

Γ	CS-CPU	CS-Accel	CS-ID	C-Tandem
Γ_1	**1728**,1135	2032,**1033**	1881,**1033**	**1728**,1135
Γ_2	**499**,1135	2032,**319**	806,421	499,523
Γ_3	**1267**,1135	2032,**115**	**1267**,727	**1267**,217
Γ_4	**346**,1135	2032,**931**	1728,**931**	653,**931**

nization approach was used to arbitrate GPU access. Tasks are allocated to cores using Sync-Aware WFD. For all our proposed techniques, we compute the frequency(ies) using our theoretical model, and then choose the next larger available CPU/GPU frequency. Our algorithms depend on the ratio K_{cpu}/K_{gpu}. In our calculations, the power-model parameters are set to: $K_{cpu} = 1$, $K_{acc} = 1$ and $\alpha = 3$. This is based on the fact that both the CPU and the GPU lie on the same chip [22]. We also assume that the task WCET scales with the frequency. The frequencies we computed for the different tasksets can be found in Table 8.3.

The energy consumption of each taskset is measured over multiple hyper-periods by using the on-board INA3221 power monitors [183]. To compute the energy consumption, we periodically read the monitors, which measure the power drawn by the 4 ARM cores and the GPU, every 10 ms. The possible sources of error in our measurements are: (i) the INA3221 has a measurement accuracy of 0.1%, and (ii) the overhead of reading the INA3221 on the CPU. However, we believe that these sources of error are small and do not qualitatively affect our results.

Table 8.4: Power Measurements on the NVIDIA TX2 (milliwatts)

Γ	Type	No VFS	CS-CPU	CS-Accel	CS-ID	C-Tandem
Γ_1	Total	1826.91	1435.20	1805.91	1601.79	**1425.93**
	CPU	1338.36	995.07	1363.23	1160.09	**985.45**
	GPU	488.03	440.12	**440.12**	441.69	440.48
Γ_2	Total	906.23	607.38	835.82	537.31	**530.12**
	CPU	607.14	**305.06**	611.41	312.01	305.59
	GPU	299.09	302.32	**224.41**	225.30	224.53
Γ_3	Total	1405.27	797.32	1363.00	784.31	**782.74**
	CPU	1211.90	636.12	1209.90	636.12	**629.53**
	GPU	193.37	161.20	**153.10**	161.20	153.20
Γ_4	Total	632.57	419.61	625.28	539.78	**404.65**
	CPU	469.41	**238.95**	467.97	381.29	244.86
	GPU	163.15	180.65	**157.30**	158.48	159.78

To get a flavor of the realized energy savings, $\Gamma_{i|i=1,2,3,4}$ have differing amounts of CPU and GPU utilization. The results are described in Table 8.4. Compared to the case without frequency scaling, CycleTandem (C-Tandem) delivers the most energy savings, and for taskset Γ_3, we observe 44.29% lower power, which translates to a 1.78x increase in the life of a battery-powered system. Similarly, when the CPU utilization is low (Γ_2 and Γ_4), using CycleSolo-CPU (CS-CPU) yields up to 32% reduction in power. Note that, as we reduce the operating frequency, we observe greater reduction in CPU energy as compared to GPU energy. This is likely because (i) the GPU energy also depends on the frequency of the GPU memory, and, (ii) since there are multiple CPU cores, there is more opportunity to reduce the CPU frequency.

8.6 Summary

In this chapter, we introduced energy-saving fixed-priority scheduling techniques for real-time systems with non-preemptive hardware accelerators. We first proposed the CycleSolo algorithms for systems where only the CPU or accelerator frequency can be set. CycleSolo utilizes our novel *ratchet search* technique to compute a tight range containing

the lowest frequency, following which a binary search in this range yields the optimal frequency, for a given schedulability-analysis technique.

We also introduced the CycleTandem algorithm for systems where the processor and accelerator frequencies can be independently set. We compute the feasible ranges containing the energy-optimal CPU and accelerator frequency pair, and propose a greedy-search heuristic to find a good solution.

Finally, we extend the CycleSolo and CycleTandem algorithms to fully-partitioned multi-core processors. In this context, we also propose the *Sync-Aware* Worst-Fit Decreasing heuristic which yields better schedulability and energy savings, than Worst-Fit Decreasing. Analytical experiments show that our proposed techniques can deliver significant energy savings. In addition, practical experiments on the NVIDIA TX2 indicate significant energy savings – up to 44.29%, and validate the effectiveness of our theoretical contributions.

Chapter 9

Co-Scheduling Real-Time Workloads on Concurrent Hardware Accelerators

Hardware accelerators often support concurrent execution, where requests from different tasks can be executed in parallel, leading to better throughput and resource utilization. This is especially true for modern GPU architectures such as NVIDIA Fermi and Pascal [49] [50]. For example, the NVIDIA Xavier [21] has 512 cores which can be utilized by concurrent kernels. Such platforms often provide in-built schedulers which aim to maximize concurrency and throughput, but do not take into account task deadlines. Recent work [110] [111] has focused on using online techniques to co-schedule multiple real-time tasks on an accelerator, so as to increase throughput and minimize deadline misses. However, to the best of our knowledge, no known framework analyzes the schedulability of tasksets being scheduled on platforms consisting of a multi-core processor coupled with an accelerator supporting concurrent execution.

Traditionally, co-scheduling task requests concurrently on hardware accelerators like GPUs has considered the *global* scheduling paradigm. In this paradigm, tasks requests are ordered in a single queue [56] and dispatched to be scheduled on any part of the global resource. However, recent GPU architectures such as NVIDIA Volta [36] coupled with software-partitioning techniques [37] have enabled GPUs to be partitioned

into multiple "fractional" components. The ability to create both compute and memory partitions can lead to increase predictability and lower interference. Therefore, in this work, we also explore the impact of hardware accelerator partitioning on taskset schedulability.

The primary contributions described in this chapter are as follows:

- We propose a schedulability-analysis framework for fixed-priority work-conserving scheduling on platforms containing a hardware accelerator supporting concurrent execution. In particular, we introduce the novel *liquefaction* and *wavefront* techniques which aid in computing a safe upper bound on the blocking duration.
- We propose a schedulability-analysis framework for non-work-conserving FIFO scheduling on a hardware accelerator supporting concurrent execution.
- We evaluate ad identify conditions where our proposed approach works better than existing analyses [53].
- We propose techniques to partition the hardware accelerator into discrete partitions, and allocate the taskset to those partitions.

9.1 Background and System Model

We now present the system model used in this chapter. We also briefly describe the synchronization-based approach used to govern access to hardware accelerators [52], along with its suspension-based schedulability analysis introduced in [53].

9.1.1 Assumptions and Task Model

Consider a taskset Γ consisting of n sporadic real-time tasks $\tau_1, \tau_2, ..., \tau_n$. The taskset is deployed on an m-core homogeneous multi-core processor M, with cores P_1, P_2,..., P_m

coupled with a single *non-preemptive* hardware accelerator A, which supports concurrent simultaneous execution of requests belonging to different tasks. We also assume that (i) each task submits its request to the accelerator and self-suspends on the CPU, and (ii) each task issues only one request at a time.

Each task $\tau_i \in \Gamma$ is characterized by: (i) C_i, the worst-case execution time (WCET) on the CPU, (ii) T_i, the period or minimum job inter-arrival time (sporadic tasks), (iii) D_i, the relative deadline from the arrival time, and (iv) η_i, the number of accelerator requests made in each job of the task. Each accelerator request $\tau_{i,k}|k = 1, 2, ..., \eta_i$ is characterized by the tuple $\{G_{i,k}, F_{i,k}, \zeta_{i,k}\}$, where $G_{i,k}$ is the WCET, $F_{i,k} \in (0, 1]$ is the fraction of the accelerator requested, and $\zeta_{i,k} \in Z^+$ is the number of accelerator segments in a request. Unless specified otherwise, we assume that $\zeta_{i,k} = 1$. The term $G_{i,k}$ consists of: (i) $G^e_{i,k}$, the WCET of the task on the accelerator, and (ii) $G^m_{i,k}$, the worst-case CPU-intervention required to access the accelerator. Note that $G_{i,k} \leq G^e_{i,k} + G^m_{i,k}$, as $G^e_{i,k}$ and $G^m_{i,k}$ may not occur on the same control path [53]. However, to model the worst case, we assume that $G_{i,k} = G^e_{i,k} + G^m_{i,k}$. For each task τ_i, the total CPU time required is $E_i = C_i + \sum_{k=0}^{\eta_i} G^m_{i,k}$, and the total accelerator time required is $G_i = \sum_{k=0}^{\eta_i} G_{i,k}$.

CPU Scheduling Policy: For tasks executing on the multi-core CPU, we consider fully-partitioned fixed-priority preemptive scheduling. Each task τ_i is assigned a unique priority π_i, such that τ_j has a higher priority than τ_i if $\pi_j > \pi_i$. $P(\tau_i)$ is used to denote the set of tasks allocated to the same core as τ_i. We use $hp(\tau_i)$ and $lp(\tau_i)$ to represent the set of all tasks which have a base priority higher and lower than τ_i respectively.

Accelerator Scheduling Policy: The accelerator can have a different scheduling policy than the CPU. In practice, most commercial accelerators do not disclose their internal scheduling policy. However, prior work indicates that GPUs execute requests in FIFO order [184]. Therefore, an accelerator is typically modeled as a shared resource, and a *lock* [52] or a *server* [51] is used to decide which task is dispatched to the accelerator. Hence, by implementing an appropriate scheduling policy in this lock or server, the order in which an accelerator receives requests can be controlled. Since we consider global

scheduling, we assume that the accelerator maintains a single queue of incoming task requests, which are dispatched to the accelerator based on (i) resource availability, and (ii) scheduling policy. Similarly, for partitioned scheduling, we assume that each accelerator partition has its own request queue, and tasks cannot migrate between partitions. Note that each partition can have a different size (accelerator fraction). Scheduling policies on the accelerator can also be characterized as work-conserving or non-work-conserving. A work-conserving scheduler always keeps a resource busy if any waiting job can execute using the available resource, while a non-work conserving scheduler can idle the resource even if there is some waiting job.

9.1.2 Schedulability-Analysis Preliminaries

To determine taskset schedulability, the response-time-based analysis is used, and the worst-case response time for a task τ_i is given by the following recurrence [53]:

$$W_i^0 = C_i + G_i + B_i, W_i^{k+1} = C_i + G_i + B_i + \sum_{h=1}^{i-1} I_{i,h} \tag{9.1}$$

where, W_i is the worst-case response time of the task τ_i, B_i provides an upper-bound on the worst-case blocking faced by τ_i in getting access to the accelerator, and $I_{i,h}$ denotes the worst-case CPU preemption τ_i faces due to a higher-priority task τ_h. If $W_i \leq D_i$, then τ_i will be schedulable.

Preemption: The worst-case preemption $I_{i,h}$ faced by task τ_i due to a higher-priority task τ_h on the CPU is given by:

$$I_{i,h} = \alpha_{i,h} * E_h, \alpha_{i,h} = \lceil (W_i + W_h - E_h)/T_h \rceil \tag{9.2}$$

where, $\alpha_{i,h}$ represents an upper bound on the number of jobs of τ_h released during a single job of τ_i [53]. Note that $\alpha_{i,h}$ considers the jitter, $W_h - E_h$, introduced by τ_h's self-suspension on the CPU, while accessing the accelerator [107].

Blocking: For accelerators which *do not* support concurrent execution, under fixed-priority non-preemptive scheduling, a synchronization protocol or a server is used to

govern accelerator access. The worst-case blocking B_i, faced by a task τ_i, in accessing such an accelerator can be upper-bounded by multiple approaches described in prior work [53] [54] [55]: (i) the job-driven analysis, (ii) the request-driven analysis, and (iii) the hybrid analysis. Neither of the first two analyses strictly dominates the other. In practice, the work in [53] observed that the job-driven analysis dominates the request-driven analysis when the number of per-job task accelerator requests increases. On the other hand, as C_i and W_i increase, the job-driven analysis becomes more pessimistic. The hybrid analysis proposed in [53] combines both analyses to provide a less-pessimistic worst-case response-time estimate.

The concurrency of the accelerator does not impact the high-priority interference that tasks face on the CPU. However, compared to non-concurrent accelerators, the blocking calculation changes. Therefore, we extend existing analyses to the context of accelerators supporting concurrent execution. In terms of accelerator scheduling policies, we consider global scheduling, and first analyze work-conserving fixed-priority scheduling. Subsequently, we analyze non-work conserving first-in-first-out (FIFO) scheduling.

9.2 Work-Conserving Fixed-Priority Scheduling

We now describe the schedulability analysis considering work-conserving fixed-priority scheduling on both the CPU and the concurrent accelerator. On the accelerator, we consider non-preemptive global scheduling, where all task requests are enqueued into a single priority queue. At any point of time, if some fraction $0 < f \leq 1$ of the accelerator is free, then the highest-priority task request waiting to run, which has a fractional requirement, $F_{i,k} \leq f$, is scheduled.

9.2.1 Concurrency-Induced Complexities

The fact that multiple requests can concurrently execute on the accelerator violates the "mutually-exclusive access" assumption of existing analysis techniques. This is in partic-

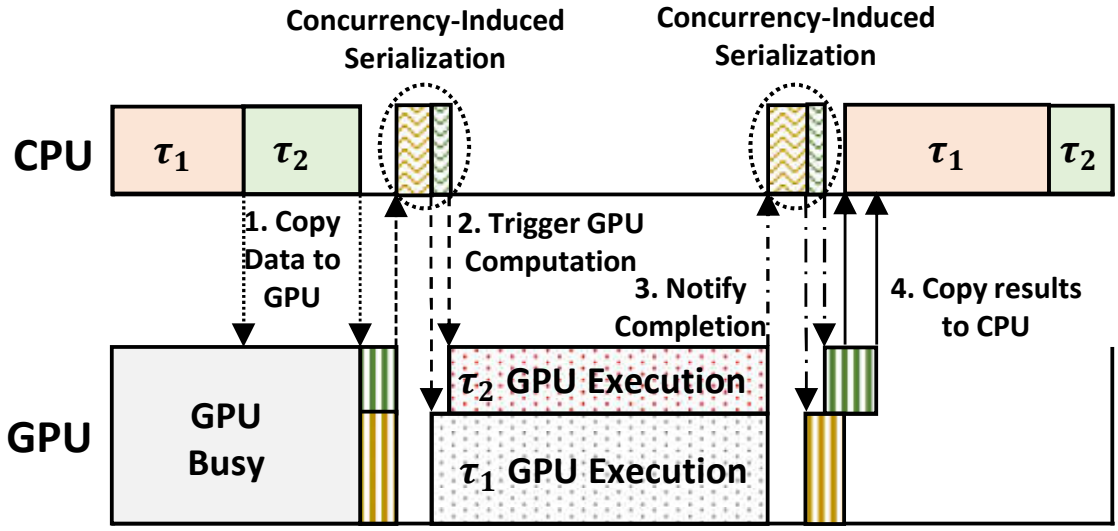


Figure 9.1: Concurrency-Induced Serialization on the CPU

ular consequential for analyses which assume the use of resource-access protocols such as the Multiprocessor Priority-Ceiling Protocol (MPCP) [105]. Additionally, if a critical section has both CPU and accelerator components (as is common in GPU requests), then requests executed concurrently on the accelerator can suffer some serialization of their CPU segments. We refer to this phenomenon as *concurrency-induced serialization*, and it is illustrated in Figure 9.1, using a GPU-access example with two tasks τ_1 and τ_2. Observe that, despite there being sufficient accelerator resources to ensure both τ_1 and τ_2 can use the accelerator, the CPU components of the accelerator requests can get serialized.

Multi-Access MPCP: We introduce a modified version of MPCP, called Multi-Access MPCP (MA-MPCP), which preserves the key properties of MPCP while (i) allowing multiple tasks to concurrently utilize a shared resource, and (ii) arbitrating access to the CPU for requests undergoing concurrency-induced serialization. Like MPCP, each shared resource R_p has a resource-specific priority queue Q_p which governs its access. In the context of concurrent-accelerator access, the key rules of MA-MPCP are:

1) A task τ_i, with priority $\pi_i \in Z^+$, requesting access to a resource R_p, with a fractional request F_i is suspended and inserted into a resource-specific priority queue.

2) When a task τ_j with a fractional request F_j completes its resource request, it releases the fraction F_j of the resource. Subsequently, the task τ_i closest to the head of the priority

queue with a fractional requirement $F_i \leq F_j$ is scheduled, and granted access to the resource. In doing so, the priority of τ_i is raised to Π_i, given by:

$$\Pi_i = \Pi_B + \Pi_h + [(\pi_i - \Pi_l)/(\Pi_h - \Pi_l + 1)] \tag{9.3}$$

where Π_B is a priority level greater than the base priority of any task [105], while Π_h and Π_l are the highest and lowest base priority of any task using the resource R respectively.

3) On resource access completion, τ_i releases the fraction F_i of the resource, and its priority returns to its base priority.

Key Observations: Compared to MPCP, on a task successfully acquiring a fraction of the resource, MA-MPCP raises the task priority to the priority ceiling, $\pi_i^C = \Pi_B + \Pi_h$, plus an additional fractional term $\pi_i^R = [(\pi_i - \Pi_l)/(\Pi_h - \Pi_l + 1)] \in [0, 1)$. This additional term has two implications:

1) Tasks holding resource R_p have a unique priority that is proportional to their base priority. Thus, for tasks concurrently executing on a resource, concurrency-induced serialization on the CPU proceeds in the original priority order.

2) It allows the raised priority of tasks holding resource R_p to be less than the raised priorities of tasks holding resource R_q with a higher priority ceiling. Thus, MA-MPCP has the same behavior as MPCP, when it comes to interference caused by tasks accessing resources with higher priority ceilings.

Therefore, building on the MPCP-based blocking analysis for non-concurrent accelerators [53], we can split up the blocking faced by tasks into the following components: (i) *direct blocking*, caused by tasks τ_j using a resource requested by τ_i, (ii) *indirect blocking*, caused when tasks τ_k accessing a resource with a higher priority ceiling preempt the execution of τ_j, which is holding a resource that τ_i is waiting for, (iii) *prioritized blocking*, which is incurred when lower-priority tasks τ_l executing their critical sections with priority ceilings preempt the CPU execution of τ_i, and (iv) *concurrency-induced serialization*, incurred when higher-priority tasks τ_h executing concurrently on the shared resource with τ_i block the CPU critical sections of τ_i. This concurrency-induced serialization does

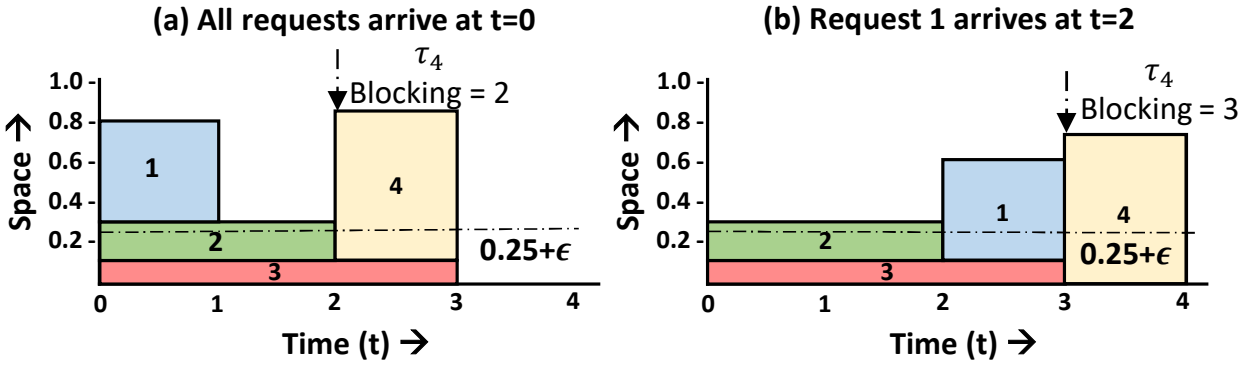


Figure 9.2: Lack of a Critical Instant for Concurrent GPUs

not occur in non-concurrent accelerators.

We now describe the analysis of each of these blocking components in the following sub-sections.

9.2.2 Direct Blocking

The introduction of concurrency changes the direct blocking calculation significantly due to the following reasons: (i) the critical instant is undefined, (ii) the blocking depends on the order in which task requests arrive and are scheduled, and (iii) some low-priority tasks which arrive with or after the blocked request can still contribute to its blocking. We look into each of these components in the context of direct blocking faced by a single request of a task, and later generalize it to a more concrete analysis framework.

Lemma 1: For concurrent accelerators using non-preemptive work-conserving fixed-priority scheduling, the worst-case blocking suffered by a request b of task τ_i *does not always* occur when all requests corresponding to other tasks arrive together with the blocked request.

Proof: We illustrate the lack of a fixed critical instant by providing an example. Consider taskset Γ, with four implicit-deadline tasks, each containing a single accelerator request. Without loss of generality, let the CPU intervention required for each accelerator request be $G^m = 0$. Given these constraints, the task execution parameters are given by (C_i, G_i, F_i, T_i), where G_i denotes the WCET on the accelerator, and F_i the ac-

celerator fraction required by a request. The taskset is as follows: $\tau_1 = (1, 1, 0.5, 10)$, $\tau_2 = (1, 2, 0.2, 20)$, $\tau_3 = (1, 3, 0.1, 30)$, $\tau_4 = (1, 1, 0.75, 40)$. Figure 9.2 illustrates the blocking faced by the lowest-priority task τ_4 under two different scenarios. Figure 9.2a shows that the blocking is 2 when all high-priority requests arrive together with τ_4's request. This is because, at time $t = 2$, 0.9 of the accelerator becomes available, which is greater than the fraction $F_4 = 0.75$ required by τ_4's request. Alternatively, Figure 9.2b shows that the blocking is 3, when the highest-priority task τ_1's request arrives later at time $t = 2$. This proves that the worst-case blocking pattern does not necessarily involve all the higher-priority requests arriving with the blocked request. ■

Key Observation: Given that we are considering *global* scheduling on the accelerator, Lemma 1 is analogous to the undefined *critical instant* in global multi-processor scheduling [185]. Therefore, to calculate a safe upper bound on the worst-case blocking faced by a task request, we need to re-frame the blocking calculation as the following optimization problem.

Objective: To maximize the blocking faced by the k^{th} accelerator request of a task τ_i, with a fractional requirement of $F_{i,k}$, find the longest valid sequence of the concurrent execution of task requests which keeps at least $1 - F_{i,k} + \epsilon$ fraction of the accelerator resource busy, for some small $\epsilon > 0$.

The optimal solution to this objective function is a sequence of task requests which keep the accelerator just busy enough to delay the execution of a blocked task request as much as possible, thus, maximizing its blocking. However, unlike the non-concurrent accelerator case, this optimal sequence and, as a consequence, the worst-case blocking depend on:

1. the order in which requests are fulfilled – which impacts which requests execute concurrently to *just* keep the accelerator sufficiently busy to block the task, and

2. the time at which requests arrive – which impacts both how tasks execute concurrently as well as how many jobs of a task can contribute to the blocking sequence.

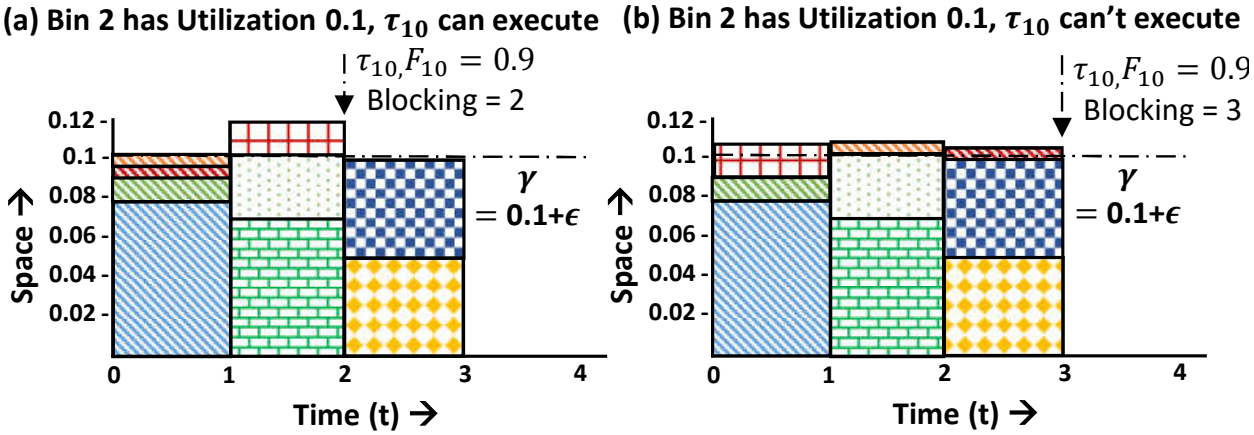


Figure 9.3: Example framing the blocking calculation as a bin-packing problem

This unfortunately results in the optimization problem becoming combinatorial in nature, which requires an exhaustive search to find the optimal solution. Consider the following example taskset with 10 tasks, each with period ∞. For the sake of simplicity, we initially assume task periods as ∞, as this avoids complications associated with new task requests arriving. We will relax this assumption in later steps. Each task has a unit-size accelerator request, and the fraction F_i of the accelerator required for each task to execute is given by the following set $\{0.08, 0.006, 0.012, 0.003, 0.07, 0.03, 0.02, 0.05, 0.05, 0.9\}$. Let us calculate the worst-case blocking faced by the lowest-priority task τ_{10}'s request with a fraction $F_{10} = 0.9$. Based on the objective function, we need to find the longest sequence which keeps at least a fraction $f_{busy} > 0.1$ of the processor busy. Figures 9.3a and 9.3b show two sequences of task request orderings which cause a blocking of 2 and 3 respectively, and in this case, 3 is the worst-case blocking duration.

Max-Resource Bin Packing: This problem is similar to the *bin-packing* [186] problem, with the objective as follows: "maximize the number of bins such that each bin has at least γ material in it". Our problem is identical to the maximum-resource bin-packing problem, which is known to be NP-complete [186]. In this case, each bin represents a unit amount of time in the blocking sequence, and $\gamma = 1 - F_{i,k} + \epsilon$, for some small $\epsilon > 0$, is the amount of material required to just *overflow* a bin, and therefore block the request under consideration from executing. In a real-world setting, the unit length of

time and the value ϵ are system constraints dictated by the smallest accelerator execution granularity in both space and time.

Key Observations: In the stated example scenario, each task had a single request which was of unit length. If these restrictions are lifted, then the bin-packing problem is augmented by two constraints: (i) as execution on the accelerator is non-preemptive, unit-length pieces corresponding to a single request need to go into adjacent bins, and (ii) multiple requests corresponding to the same task cannot go into the same bin, as each task request must complete before the next request can be enqueued. This is in effect a more constrained form of the stated bin-packing problem, and is hence also NP-complete. Therefore, we seek a polynomial algorithm that *always* yields a safe upper bound on the blocking. Note that this solution should be greater than or equal to the optimal number of bins to be safe.

Theorem 1: The *optimal* solution to the maximum-resource bin-packing problem *always* yields a number of bins which is *greater than or equal to* the optimal solution to the constrained version described above.

Proof: For a taskset Γ, assume that the optimal solution to the constrained version of the maximum-resource bin-packing problem yields more bins than the optimal solution to its unconstrained version considering unit-sized (in time) pieces of task requests. However, the optimal solution to the constrained problem should also be a valid optimal solution to the un-constrained problem. Hence, there is a contradiction. ■

Theorem 1 implies that finding a safe upper bound to the number of bins yielded by the optimal solution to the maximum-resource bin-packing problem will yield a safe estimate of the worst-case blocking. Note that serializing all the task requests is one such *safe* upper bound. However, it is overly pessimistic and yields a blocking estimate similar to the non-concurrent case. Therefore, we require an estimate that is safe but still utilizes the concurrency of the accelerator. One such solution we propose is *liquefaction*.

Liquefaction

Liquefaction involves breaking up or *liquefying* each task request $\{G, F\}$ into multiple smaller component requests with parameters $\{G', F'\} = \{1, \epsilon\}$, for some small $\epsilon > 0$. The optimal solution to the maximum-resource bin-packing problem, for these transformed uniform-sized *liquefied* components, can be easily found by *pouring* the liquid components into unit-width (in time) bins (or jugs) of height γ, where $\gamma = 1 - F_{i,k} + \epsilon$, for some small $\epsilon > 0$, is the amount of material required to just *overflow* the bin, and block the request $\tau_{i,k}$ under consideration from executing. Each time a bin overflows, a new bin is opened until we run out of "liquid". The number of overflowing bins is the optimal solution for this new transformed taskset.

Theorem 2: Consider n blocking requests, each with duration G_i, and fractional requirement F_i, which block a request b with fractional requirement F_b. Then, liquefaction yields a safe upper bound $\Theta = \lfloor (\sum_{i=0}^{n} G_i * F_i)/\gamma \rfloor$ to the optimal number of bins for the maximum-resource bin-packing problem, where $\gamma = 1 - F_b + \epsilon$, for some small $\epsilon > 0$.

Proof: We prove the theorem using contradicition. Combining all the n liquefied requests yields $\sum_{i=0}^{n} G_i * F_i$ volume of liquid. Therefore, if each bin has a maximum size of $\gamma = 1 - F_b + \epsilon$, then the maximum number of full (or overflowing) bins is given by $\Theta = \lfloor (\sum_{i=0}^{n} G_i * F_i)/\gamma \rfloor$, where the floor function discards the last partially-filled bin. Now, suppose that the optimal solution to the maximum-resource bin-packing problem considering the original non-liquefied request is $\Theta + k$, where $k \in Z^+$. Thus, as each bin is at least filled up to the level γ, the total volume of the non-liquefied tasks must be at least $(\Theta + k) * \gamma$, simplified as: $(\Theta + k) * \gamma = (\Theta + 1) * \gamma + (k - 1) * \gamma > (\Theta + 1) * \gamma > \sum_{i=0}^{n} G_i * F_i$ which is greater than the total volume of the liquefied tasks. However, liquefaction does not change the volume of the set of requests. Hence, there is a contradiction. ■

Figure 9.4 illustrates liquefaction using a taskset with four implicit-deadline tasks, each with a single accelerator request. Without loss of generality, the CPU intervention

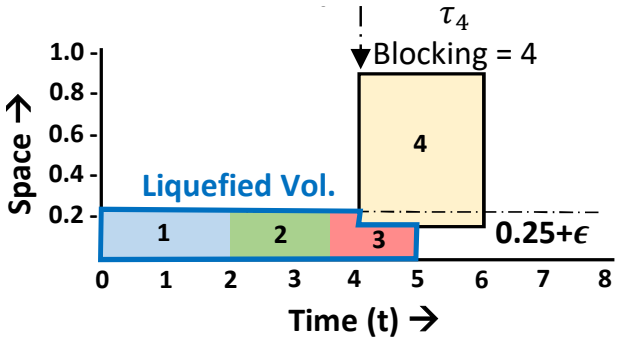


Figure 9.4: Liquefaction Example

required for each accelerator request is set to $G^m = 0$. The task parameters are given by (C_i, G_i, F_i, T_i) and, the taskset in priority-order is: $\tau_1 = (1, 5, 0.1, 10)$, $\tau_2 = (1, 2, 0.2, 20)$, $\tau_3 = (1, 3, 0.1, 30)$, $\tau_4 = (1, 1, 0.75, 40)$. We calculate the worst-case blocking for the lowest-priority task τ_4, and liquefy each task such that the "liquid" in each unit-sized time bin is filled up to the overflow point $\gamma = 1 - 0.75 + \epsilon = 0.25 + \epsilon$. Since the last bin ($t = 4$ to 5) is partially filled, there is sufficient fraction $(>= 0.75)$ for τ_4's request to execute, and hence its direct blocking is 4.

Key Observations: Liquefaction can be used to get a safe upper bound on the blocking estimate. Furthermore, while finding the optimal worst-case blocking for a request using the maximum-resource bin-packing problem has non-polynomial complexity, liquefaction yields a valid upper bound by using simple arithmetic. In addition, we can also apply some optimizations to make the bound returned by liquefaction less pessimistic, and these are given by the following results.

The **Liquefaction Factor** is the smallest fractional unit-sized chunk into which requests are broken into. For example, using a liquefaction factor of $\delta > 0$ converts a task request $\{G, F\}$ into $F * G/\delta$ smaller unit-sized fractional component requests with parameters $\{G', F'\} = \{1, \delta\}$.

Lemma 2: Consider n blocking requests, with duration G_i, and fractional requirement F_i, which block a request b. Then, liquefying the requests using the greatest common divisor $GCD_{i=0}^{n}(F_i)$ as the *liquefaction factor* yields a valid upper bound on the blocking

faced by request b.

Proof: Consider the n requests, each broken into G_i pieces with parameters $\{1, F_i\}$. If we place some combination of these pieces into unit-sized bins, the height of the bin will always be a multiple of $GCD_{i=0}^{n}(F_i)$, as the GCD perfectly divides all F_i. Thus, any combination of these pieces just reaching or exceeding γ will also be a multiple of the GCD. Therefore, while liquefying using the GCD as the liquefaction factor, stacking the liquefied *chunks* in each bin would yield a height $p * GCD$, where $p \in Z^+$ is the smallest positive integer which satisfies $p * GCD \geq \gamma$. This implies that, in the optimal solution to the maximum-resource bin-packing solution, each bin must have a height $h \geq p * GCD$. Therefore, as liquefaction conserves the volume of all the requests, and its bin height is always less than or equal to the optimal solution, it will always yield a number of bins greater than or equal to the optimal solution, yielding a safe bound on the worst-case blocking. ■

Lemma 3: Consider n blocking requests, each with duration G_i, and fraction F_i, which block a request b with fraction F_b. Then, only liquefying requests with $F_i < \gamma = 1 - F_b + \epsilon$, and assigning all other requests their own bin yields a valid upper bound on the blocking faced by request b.

Proof: In the optimal solution to the maximum-resource bin-packing problem, requests with fractional requirement $F_i \geq \gamma$ will always go into their own separate bin (or bins based on their length in time). Therefore, the blocking calculation problem can be re-phrased as a combination of two independent components: the serialized length of all requests with fractional requirement $F_i \geq \gamma$, and the optimal solution to the maximum-resource bin-packing problem considering only requests with fractional requirement $F_i < \gamma$. Therefore, from Theorem 2, liquefaction can be used to yield a safe upper bound to the optimal solution to the maximum-resource bin-packing problem considering only requests with fractional requirement $F_i < \gamma$. Combining this value with the serialized length of all requests with fractional requirement $F_i \geq \gamma$ yields a safe upper bound on the worst-case blocking. ■

Generalization to Sporadic Tasks: So far, we have only considered aperiodic tasks with period $T_i = \infty$. However, requests that arrive when a task is being blocked also contribute to direct blocking. In this scenario, liquefaction still provides a valid upper bound *if* we consider that all other task requests arrive together with the blocked request. This is despite the critical instant for calculating the worst-case blocking being undefined for concurrent accelerators (Lemma 1).

Theorem 3: Consider n sporadic tasks which can block request b requiring an accelerator fraction F_b. Then, liquefaction yields a safe upper bound on b's worst-case blocking, if we consider that all the blocking requests arrive together with b.

Proof: Let us assume that the optimal solution using liquefaction yields a safe upper bound on the blocking with Θ bins. Assume that this optimal solution includes at least one task τ_j's first blocking request which arrives before or after the request b. Therefore, the total volume of all tasks contributing to blocking is at most $M = \Theta * \gamma + (\gamma - \epsilon)$, where $\gamma = 1 - F_b + \epsilon$. Now, if all requests arrive with the blocked request, let the total volume be $M' < M$, which yields a solution with bins Θ' lesser than or equal to the optimal solution Θ. However, there is a contradiction, as all tasks arriving together with the blocked request b allows the maximum jobs of a task to be considered and yields the maximum volume. This follows from the critical-zone theorem for fixed-priority uniprocessor scheduling [170] [29]. ■

Therefore, liquefaction effectively converts the multi-faceted bin-packing-based problem into one of maximizing the amount of blocking resource and using it to fill up space. However, while the bound is valid, liquefaction can still be pessimistic.

Blocking Tasks: While liquefaction is useful, we still need to establish which tasks accessing the accelerator contribute to blocking. Due to the nature of fixed-priority scheduling, a task can be blocked by all higher-priority requests. However, utilizing work-conserving scheduling also implies that if a processor is idle and no higher-priority tasks can run (due to an insufficient available fraction), then some lower-priority tasks can jump the queue and contribute to worst-case blocking.

Lemma 4: For concurrent accelerators, a task τ_b's request with fractional requirement F_b can also be blocked by lower-priority requests l, requiring a smaller fraction $F_l < F_b$ of the accelerator, which arrive with or after the blocked request.

Proof: Consider a scenario where a blocked task request corresponding to τ_b is blocked by higher-priority task requests corresponding to tasks $\tau_h | h \in hp(\tau_b)$. Let the available accelerator fraction be $F_b - \epsilon$, for some small $\epsilon > 0$. Now, if all the higher-priority tasks have a fraction $F_h > F_b$, then neither tasks τ_h or τ_b can execute on the accelerator. However, as the scheduler is work-conserving, if any lower-priority task $\tau_l | l \in lp(\tau_b)$ arrives with a request with fractional requirement $F_l \leq F_b - \epsilon < F_b$, it can jump the queue and execute on the accelerator. Thus, it can block a request from τ_b. ■

The ability of lower-priority requests to jump the queue can significantly increase blocking by causing a cascading effect which allows other lower-priority tasks to jump ahead as well.

Fraction-Inverse-Monotonic Policy (FIM): One alternative to avoid this increase in blocking is by assigning priorities to requests in inverse proportion to the fraction of the accelerator they require, i.e, for the j^{th} request of τ_i, the priority $\pi_{i,j} \propto 1/F_{i,j}$. This ensures that requests with smaller fractions have a higher priority. Thus, they cannot be blocked by any lower-priority request as they always require a bigger fraction.

Lemma 5: A task τ_i's request b with fractional requirement F_b can only be blocked by requests from at most one job of a lower-priority task τ_l, if τ_l has at least one request with fractional requirement $F_l > F_b$.

Proof: From Lemma 4, only lower-priority requests with fractional requirement less than F_b, can execute ahead of b. Therefore, if even one request of lower-priority task τ_l has a fraction $F_l > F_b$, it cannot execute before b. Hence, subsequent jobs of τ_l cannot be enqueued until b completes. ■

Key Observations: Consider a request b with fractional requirement $F_b = 0.5$. Let there be a low-priority task τ_l with 6 consecutive requests each with their parameters $\{G_{l,j}, F_{l,j}\}$ given by the set {{2,0.8},{3,0.4},{4,0.3}, {1,0.7},{4,0.2},{5,0.4}}. As a conse-

quence of Lemma 5, if we consider blocking due to low-priority task τ_l's requests which arrive after b, then it can only be blocked by one job of τ_l and only by requests $j = 2, 3, 5$ and 6. Additionally, for τ_l, requests 1 and 4 both have fractional requirements greater than 0.5, and hence cannot block b by executing ahead of it. As a consequence, b can only be blocked by requests 2 and 3, or 5 and 6. Therefore, for a low-priority task τ_l, a request b can only be blocked by a consecutive sequence of requests with fractional requirement less than F_b. If we use liquefaction, then to maximize the worst-case blocking estimate of b, we need to consider the sequence of such requests corresponding to τ_l which have the greatest volume when liquefied. In the example, to maximize blocking, we should use consecutive requests 5 and 6 as they have a combined volume of 2.8, compared to requests 3 and 4 which have a volume of 2.4.

As the accelerator is non-preemptive, apart from tasks which arrive with or after a blocked request, requests already executing on the accelerator also contribute to direct blocking.

Wavefront

There is a pattern to the worst-case blocking due to already-executing requests, and we capture this notion with a "wavefront".

Theorem 4: Assume a blocked request b arrives at time t. The worst-case blocking on request b with fraction F_b caused by already-executing task requests occurs when these requests start executing at $t - \epsilon$ for some small $\epsilon > 0$. The requests which make up this pattern include the k longest requests each starting at $t - \epsilon$ and together utilizing a fraction $f > \gamma = 1 - F_b + \epsilon$.

Proof: To maximize the blocking caused by already-executing task requests, we need to keep at least γ of the processor occupied for as long as possible after a blocked request arrives. The latest a request can start to count as already executing is $t - \epsilon$, for some small $\epsilon > 0$. Therefore, the latest start ensures that each already-executing request can block request b for the longest time possible. In terms of requests, choosing the k longest

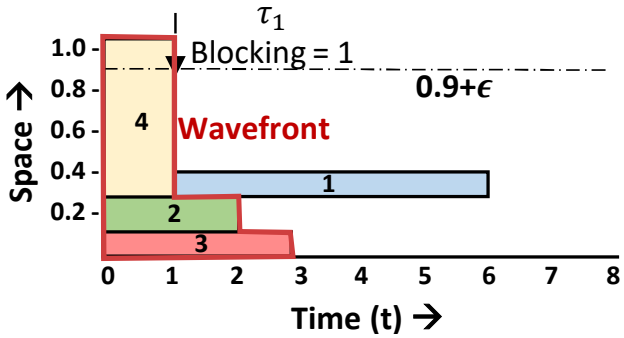


Figure 9.5: Wavefront example

requests, each starting at $t - \epsilon$ and together utilizing a fraction $f > \gamma$, ensures that at least γ of the processor is busy till the length (WCET) G_k of the k^{th} longest request. Replacing any of these requests with a shorter request will cause this busy duration to only decrease. ■

Figure 9.5 illustrates the wavefront using the same example taskset with four implicit-deadline tasks, used to illustrate liquefaction. We calculate the worst-case blocking for the highest-priority task τ_1, and construct a wavefront up to the overflow point $\gamma = 1 - 0.1 + \epsilon = 0.9 + \epsilon$. Since the bin is partially filled from $t = 1$ to 2, there is sufficient fraction left $(>= 0.1)$ for τ_1's request to execute, and hence its direct blocking is 1.

Key Observation: We consider all higher-priority and some lower-priority requests in the blocking calculation (liquefaction) as arriving with or after the blocked request b. Therefore, only requests not used in liquefaction should be used to create the worst-case wavefront of already-executing requests. For a request $\tau_{i,j}$, we define the set of requests which are a part of the wavefront using the set $\omega(\tau_{i,j})$

Estimating Direct Blocking: Combining Wavefront and Liquefaction

Consider the *wavefront* pattern created by stacking the k task requests creating the worst-case direct-blocking pattern caused by already-executing tasks on a request b. Let the set of these requests in increasing order of time duration G_i be given by the set $\Lambda_b = (\{G_1, F_1\}, .., \{G_k, F_k\})$. Thus, without loss of generality, if we assume request b arrived at $t = 0$, then requests arriving subsequently which contribute to blocking can start

executing at the time instants given by the set $\lambda_b = 0 \cup G_i \in \Lambda_b$. Therefore, we can perform liquefaction on the tasks arriving subsequently by considering the size of the first G_k bins to be $\kappa_i = max(0, \gamma_b - \sum_{j=\nu(i)}^{k} F_i)$, where $i \in [1, G_k]$ is the bin number and $\nu(i)$ is a function which returns the smallest $G_i \in \lambda_b | i \leq G_i$. All bins after G_k are of size $\gamma_b = 1 - F_b + \epsilon$.

Lemma 6: Liquefying all the requests that are part of the wavefront pattern still yields a safe direct blocking estimate.

Proof: Consider the case where the volume M from liquefying requests which arrive with or after the blocked request is not sufficient to fill the first G_k bins each of size κ_i, i.e., $M < \sum_{i=1}^{k} \kappa_i$. In this case, the direct blocking B_b^{dr} faced by b will be less than G_k. Therefore, all the volume from the wavefront that lies beyond B_b^{dr} will not contribute to the estimate. Thus, if we liquefy the requests in the wavefront, all the volume gets counted in the blocking estimate which leads to either the same or a more pessimistic but valid estimate. ■

We can now state the worst-case direct blocking computations using the request-driven, job-driven and hybrid analyses.

Request-driven Analysis: The j^{th} accelerator request of task τ_i, in the worst case, can be directly blocked for a duration $B_{i,j}^{dr}$. Then, using the request-driven analysis [53] [55] the total direct blocking can be given by: $B_i^{dr} = \sum_{j=1}^{\eta_i} B_{i,j}^{dr}$.

Using the request-driven analysis, the worst-case direct blocking faced by the j^{th} accelerator request $\tau_{i,j}$ of task τ_i can be computed using Algorithm 15. For the j^{th} accelerator request of task τ_i with fractional requirement $F_{i,j}$, direct blocking can be caused by the following independent components: (i) all higher-priority task requests, and lower-priority task requests with fractional requirement $F < F_{i,j}$, given by the set $\Phi(\tau_{i,j})$, which arrive with request $\tau_{i,j}$ and while $\tau_{i,j}$ is being blocked, and (ii) lower-priority task requests not considered above, which are already executing on the accelerator when a request arrives. As both these components occur independently, direct blocking as a result of case (i) can be computed using liquefaction (Theorem 3), and case (ii) can be computed using

Algorithm 15 Request-Driven Direct Blocking Calculation

1: **procedure** RequestDriven-Direct($\tau_{i,j}$)
2: $B' = 0, \gamma = 1 - F_{i,j} + \epsilon, \Delta = \Delta' = 0$
3: /* blocking pattern due to already-executing tasks */
4: λ, Λ = Wavefront(i, j) ▷ get wavefront requests
5: **for** $t = 0$ to $\max(H_q) \in \lambda$ **do**
6: $\nu(t) = \min(H_l \in \lambda | t \leq H_l)$
7: $\kappa_t = \max(0, \gamma - \sum_{p=\nu(t)}^{k} F_p)$ ▷ get bin size
8: $\Delta' = \Delta' + \kappa_t$ ▷ get liquefying volume used
9: $\Delta =$ GetLiquefactionVol(i, j, t)
10: **if** $\Delta < \Delta'$ **then** ▷ if insufficient volume
11: $B^{dr}_{i,j} = \max(0, t-1)$, **return** $B^{dr}_{i,j}$
12: /* blocking pattern due to arriving tasks */
13: **while** B' not equal to B **do**
14: $B' = B, \Delta = 0, B_{lp} = \max(H_q) \in \lambda$
15: **for** $\tau_h \in hp(\tau_i) \cup F_{h,k} < F_{i,j}$ **do**
16: $\beta_{i,h} = \lceil (B' + W_h - E_h)/T_h \rceil$
17: $\Delta = \Delta + \beta_{i,h} * H_{h,k} * F_{h,k}$
18: $B^{dr}_{i,j} = B = B_{lp} + \lfloor (\Delta - \Delta')/\gamma \rfloor$
19: **return** $B^{dr}_{i,j}$ ▷ request direct blocking

the wavefront (Theorem 4). Algorithm 15 uses a combination of the wavefront and liquefaction techniques to calculate the blocking. Note that, the term $\beta_{i,h}$ in line 4, captures the number of jobs which arrive while the request $\tau_{i,j}$ is being blocked, while taking self-suspensions into account. Additionally, instead of using the WCET of each accelerator request $G_{p,q}$, we instead use the worst-case response time of each request $H_{p,q}$ [53]. This captures the effects of indirect blocking and concurrency-induced serialization, as described in sections 9.2.3, 9.2.5 and 9.2.6.

Instead of using Algorithm 1, we can also obtain a valid, albeit more pessimistic bound by liquefying the requests in the wavefront ω (Lemma 6). This can be obtained using a low-complexity recurrence useful for online admission control:

$$B^{dr}_{i,j} = \left\lfloor \frac{\sum_{\tau_{l,k} \in \omega(\tau_{i,j})} H_{l,k}.F_{l,k} + \sum_{\tau_h \in \Phi(\tau_{i,j})} \beta_{i,h}.H_{h,k}.F_{h,k}}{\gamma_{i,j}} \right\rfloor$$

Note that, the floor term is used to discard the liquefied volume placed in the last partially-filled bin.

Job-driven Analysis: The job-driven approach considers the blocking that a job of a task τ_i can face. All requests corresponding to a single job of a task τ_i can suffer direct blocking caused by the following independent components: (i) all higher-priority task requests, and lower-priority task requests with fractional requirement $F_l < F_{i,j}^{max} = \max_{j \leq \eta_i}(F_{i,j})$, given by the set $\Phi(\tau_i)$, which arrive with the request and while the request is being blocked, and (ii) lower-priority task requests not considered above, which are already executing on the accelerator when each of τ_i's η_i requests arrives. Note that, using a strict interpretation, the job-driven approach does not consider individual request characteristics. Therefore, the worst-case blocking faced by the task is given by:

$$B_i^{dr} = \left\lfloor \frac{\eta_i . \sum_{\tau_{l,k} \in \omega(\tau_i)} H_{l,k}.F_{l,k} + \sum_{\tau_h \in \Phi(\tau_i)} \alpha_{i,h}.H_{h,k}.F_{h,k}}{\gamma_i} \right\rfloor$$

where, $\alpha_{i,h} = \lceil (W_i + W_h - E_h)/T_h \rceil$ represents the maximum number of jobs of τ_h which can arrive during the response time of τ_i, while taking into account the effect of self suspensions [106]. Consider the numerator of the floor function. Observe that the first term does not consider the *wavefront* pattern, as the job-driven analysis does not perform per-request liquefaction. Therefore, from Lemma 6, we instead liquefy all the tasks in the wavefront pattern $\omega(\tau_i)$, and consider η_i invocations for each request in a job of task τ_i. Subsequently, the second term performs liquefaction considering all the jobs of tasks $\tau_h \in \Phi(\tau_i)$ which arrive during the response time of task τ_i. Given that $\Phi(\tau_i)$ considers all requests with a smaller fraction than τ_i's request with the largest fraction $\max_{j \leq \eta_i}(F_{i,j})$, the job-driven analysis can yield a pessimistic analysis. In theory, if the largest request requires the entire accelerator, then the analysis will consider almost all accelerator requests. Therefore, we can make the job-driven analysis less pessimistic by considering requests of the blocked task τ_i one at a time.

Key Observation: Consider τ_i, with η_i requests, such that all requests have the same priority. Then, using the wavefront pattern with liquefaction results in requests with a bigger fractional requirement $F_{i,j}$ getting blocked for a longer duration. The reason for

Algorithm 16 Request-Oriented Job-driven Direct Blocking Calculation

procedure RO-JobDriven-Direct(τ_i)
  $B = 0, \Delta = \Delta' = 0$ ▷ Initialize Blocking
  for $j = 1$ to η_i **do**
    Δ' = GetLiquefactionVol(Wavefront(i, j))
    $\gamma_{i,j} = 1 - F_{i,j} + \epsilon$
    if $F_{i,j} < \max_{k \in [j,\eta_i]}(F_{i,k})$ **then**
      $B = B + \lfloor \Delta' / \gamma_{i,j} \rfloor + H_{i,j}$
    else
      $B =$ PerformRecurrence($i, j, \gamma_{i,j}, \Delta, B$) + $H_{i,j}$
      $\Delta = \Delta +$ GetLiquefactionVol(i, j, B)
  return $B_i^{dr} = B$ ▷ ro job-driven direct blocking

this is twofold: (i) a smaller fraction of the accelerator needs to be kept busy to block requests with bigger $F_{i,j}$, and (ii) more tasks are considered in the blocking calculation for liquefaction. Therefore, considering individual requests with smaller fractional requirement $F_{i,j}$ in the job-driven analysis can yield a less pessimistic blocking estimate.

Request-oriented (RO) Job-driven Analysis: Like the simple job-driven analysis, its request-oriented version considers all the jobs of blocking tasks which arrive during a job's response time. However, instead of considering the biggest request of the blocked job, it considers specific requests.

Key Observation: Consider a job of task τ_i with η_i requests. Suppose that τ_i's requests are ordered such that $F_{i,1} < F_{i,2} < ... < F_{i,\eta_i}$. In this scenario, in the worst case, each request can be blocked by task requests already executing, which for request $\tau_{i,j}$ can be upper-bounded by liquefying the wavefront set $\omega(\tau_{i,j})$. Additionally, each request can also be blocked by all the jobs of other tasks which arrive during τ_i's response time. However, the blocking pattern which leads to maximal blocking occurs when all these requests end up blocking the request F_{i,η_i} with the maximum fractional requirement. Alternatively, if the request $F_{i,k}|k < \eta_i$ has the biggest fractional requirement, then we could perform the liquefaction recurrence calculation considering that request until convergence. Subsequently, of the remaining requests, we can choose the next biggest request, and so on until we reach the last request. Note that we do not need to

consider jobs already accounted for in previous requests. Algorithm 16 describes this request-oriented job-driven blocking calculation.

Hybrid Analysis: The hybrid analysis uses a combination of the job-driven and request-driven approaches to yield a less-pessimistic worst-case blocking estimate. If we consider a simple case, then we can return the worst-case blocking as the minimum of that returned by both the job-driven and request-driven approaches. This is a valid bound as both bounds are valid. Note that, in the non-concurrent accelerator scenario, if we consider blocking due to already-executing tasks, the hybrid analysis considered the η_i longest lower-priority requests which arrive during the response time of the task [53]. However, for concurrent accelerators, we need to consider the wavefront pattern. Hence, limiting the number of some requests to the number of jobs which arrive during the response time, along with building per-request wavefronts, makes it a combinatorial problem. Therefore, for the concurrent accelerator case, the worst-case direct blocking faced by task τ_i can be stated as:

$$B_i^{dr,hybrid} = \min(B_i^{dr,job}, B_i^{dr,request}) \tag{9.4}$$

9.2.3 Indirect Blocking

Indirect blocking is caused when a task τ_k accessing a resource with a higher priority ceiling preempts the CPU critical section execution of τ_j, which is holding a resource that τ_i is waiting to access. This occurs when τ_k is scheduled on the same CPU core as τ_j. Note that, (i) indirect blocking occurs due to execution on a different resource, which can be another accelerator or a partition of the same accelerator and (ii) it occurs due to CPU segments of different critical sections interfering. Therefore, it is not affected by concurrency, and the indirect blocking faced by a request of task τ_i is identical to the non-concurrent accelerator case as described in [53].

Lemma 7: The worst-case indirect blocking faced by the p^{th} critical section of τ_j is

given by [53]:

$$B^{ir}_{j,p} = (\zeta_{j,p} + 1) * (\sum_{\tau_q \in P(\tau_j)} \max_{\pi_{q,k} > \pi_{j,p}} (G^m_{q,k})) \tag{9.5}$$

9.2.4 Prioritized Blocking

Prioritized blocking is caused by lower-priority tasks τ_l executing the CPU segments of their critical sections at the priority ceiling of the resource, as a result of using MA-MPCP. This execution can preempt the CPU execution of τ_i scheduled on the same CPU as τ_l. Note that prioritized blocking is caused by CPU segments of lower-priority critical section execution interfering with non-critical section CPU segments of higher-priority tasks. Therefore, it is not affected by concurrency and the prioritized blocking faced by a request of task τ_i is identical to the non-concurrent accelerator case as described in [53]. Using the analysis in [53], the prioritized blocking faced by a task τ_i, B^{pr}_i, can be computed using the request-driven, job-driven and hybrid analysis.

9.2.5 Concurrency-Induced Serialization

Concurrency-induced serialization is unique to concurrent accelerators. Consider two tasks τ_h and τ_i assigned to the same CPU core, both making requests to the same concurrent accelerator. When both τ_h and τ_i execute requests concurrently on the shared resource, the CPU segments corresponding to their critical sections can get serialized on the CPU. Using MA-MPCP (Section 9.2.1) assigns unique raised priorities to tasks executing their critical sections. These raised priorities are assigned in proportion to their base priorities. As a result, only a critical section corresponding to higher-priority tasks τ_h can block the CPU segment of the critical section of τ_i. Note that every request p of τ_i with fractional requirement $F_{i,p}$ can only suffer concurrency-induced blocking by higher-priority requests which have a fractional requirement $F \leq 1 - F_{i,p}$.

Lemma 8: The worst-case concurrency-induced serialization incurred by the p^{th} crit-

ical section of τ_i is given by:

$$B_{i,p}^{cis} = (\zeta_{i,p} + 1) * (\sum_{h \in hp(\tau_i) \wedge P(\tau_i)} \max_{\substack{k \leq \eta_h \wedge \\ F_{h,k} \leq 1 - F_{i,p}}} (G_{h,k}^m)) \tag{9.6}$$

Proof: The term $\zeta_{i,p}$ captures the number of accelerator accesses made in one critical section for request p of τ_i. As there will be $\zeta_{i,p} + 1$ CPU segments required to facilitate accelerator access in the critical section, each of these segments can be preempted by higher-priority tasks $\tau_h | h \in hp(\tau_i)$ on the same CPU core as τ_i, which concurrently access the resource. Additionally, as we cannot utilize more than 100% of the accelerator, only requests with fractional requirement $F_{h,k} \leq 1 - F_{i,p}$ can execute concurrently on the accelerator with $\tau_{i,p}$. The term $\max(G_{h,k}^m)$ captures the request of task τ_h with the longest CPU critical section intervention. Each such task τ_h, after accessing a critical resource, will return to its base priority. Therefore, only one request of each task τ_h can contribute to concurrency-induced serialization. ■

9.2.6 Putting it All Together

The work in [53] captures indirect blocking on each critical section instance of τ_j and incorporates it into the analysis through its impact on direct blocking. Therefore, for the p^{th} critical section of task τ_j, the term $H_{j,p}$, which captures the worst-case response time, is used in the direct blocking analysis instead of the WCET $G_{j,p}$. However, for concurrent accelerators, apart from indirect blocking, the CPU segments of the critical section can also face concurrency-induced serialization. Hence, from Lemmas 7 and 8, the worst-case response time for the p^{th} critical section of τ_j is given by:

$$H_{j,p} = G_{j,p} + B_{j,p}^{ir} + B_{j,p}^{cis} \tag{9.7}$$

Theorem 5: The worst-case total blocking B_i faced by task τ_i while accessing a concurrent accelerator can be upper-bounded by $B_i = B_i^{dr} + B_i^{pr} + B_i^{ir} + B_i^{cis}$, where B_i^{dr} is the worst-case direct blocking, B_i^{pr} is the worst-case prioritized blocking, B_i^{ir} is the worst-case indirect blocking and B_i^{cis} is the worst-case concurrency-induced serialization.

Proof: The proof follows from Theorem 1 in [53]. However, in [53], the authors do not consider the effects of indirect blocking on τ_i's accelerator execution itself. Therefore, we consider both the indirect blocking and concurrency-induced serialization faced by τ_i in the total blocking calculation. As stated in Equation 9.7, the worst-case indirect blocking and concurrency-induced serialization suffered by other requests, while they block τ_i, are considered in direct blocking. ■

9.3 Non-Work-Conserving FIFO Scheduling

We now describe the schedulability analysis considering work-conserving fixed-priority scheduling on the CPU and non work-conserving FIFO (First-In-First-Out) scheduling on the accelerator. Requests are scheduled on the accelerator in order of their arrival, i.e., a request $\tau_{i,j}$ with fractional requirement $F_{i,j}$ is scheduled as soon as (i) all requests which arrived before $\tau_{i,j}$ have been dispatched to the accelerator, and (ii) a fraction $F \geq F_{i,j}$ of the accelerator is available.

As the CPU uses fixed-priority scheduling, the CPU segments of the accelerator requests are not scheduled using the FIFO policy. Therefore, when a task request $\tau_{i,j}$ accesses the accelerator, we raise its priority to the value as prescribed by the MA-MPCP protocol in Section 9.2.1. As a result of this choice, the worst-case indirect blocking, prioritized blocking and concurrency-induced serialization incurred by task requests remain the same as compared to the fixed-priority scheduling case. However, the direct-blocking calculation changes, as given by the following theorem.

Theorem 6: For FIFO scheduling, in the worst case, each accelerator request $\tau_{i,j}$ can be directly blocked by at most one request of all the other tasks τ_k accessing the accelerator.

Proof: Let $\tau_{i,j}$ be blocked by more than one request of a task τ_k. For each task τ_k, an accelerator request must complete before the next request is enqueued. Therefore, two requests of a single task τ_k cannot be enqueued before $\tau_{i,j}$. ■

In theory, this property of FIFO scheduling can sometimes lead to better schedula-

bility than fixed-priority scheduling. This is especially true for tasksets with few tasks, with each task consisting of multiple short accelerator requests. However, when each task has multiple requests, choosing the best set of per-task requests as well as their ordering is required to create the worst-case blocking pattern. This is a combinatorial problem with no obvious solution. For a request $\tau_{i,j}$, a valid upper bound on the worst-case direct blocking, which does not consider concurrency, can be given by the sum of the longest task requests of each task: $B^{dr}_{i,j} = \sum_{\tau_p | p \neq i} \max(H_{p,q})$, where, $H_{p,q}$ (Equation 9.7) is the worst-case response time of each request. Therefore, using the request-driven approach the direct blocking faced by τ_i is given by $B^{dr}_i = \sum_{j=1}^{\eta_i} B^{dr}_{i,j}$.

Similarly, we can also calculate the direct blocking by extending the non-concurrent job-driven and hybrid approaches to FIFO. However, if we consider concurrency, the problem is again combinatorial. The total blocking B_i faced by τ_i can still be computed using Theorem 5, $B_i = B^{dr}_i + B^{pr}_i + B^{ir}_i + B^{cis}_i$.

9.4 Concurrent Accelerator Partitioning

Partitioning an accelerator can lead to more predictable execution by reducing interference. Therefore, we now consider accelerators which support software partitioning of compute resources [35], and propose preliminary techniques to efficiently partition the accelerator. Note that memory partitioning is also required to reduce memory interference [37]. However, we only consider techniques for compute partitioning, and leave memory partitioning for future work.

Consider an accelerator A, which can be partitioned into fractional compute partitions $v_k | k \in \mathbb{N}$, each with a fractional size of f_k, such that $f_{min} \leq f_k \leq 1$, where f_{min} is an accelerator-specific constraint specifying the minimum fractional size of a partition. We assume that f_{min} also constrains the maximum number of accelerator compute partitions to $\lfloor 1/f_{min} \rfloor$. In practice, a software-partitioning framework like NVIDIA MPS [35], or a restriction on the number of memory partitions [37], may place further constraints

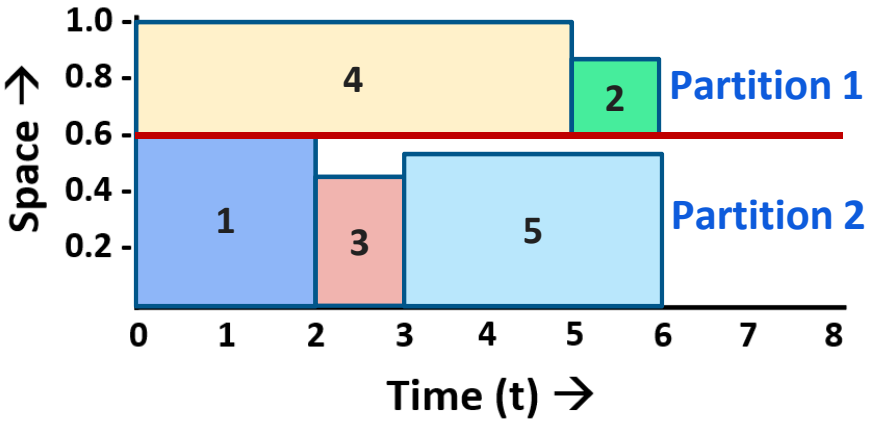


Figure 9.6: Accelerator-Partitioning Example

on the maximum number of compute partitions. We also assume that only accelerator requests with fractional requirements $F_{i,j} \leq f_k$ can be assigned to partition v_k, and requests assigned to a single partition cannot execute concurrently. This is in line with the partitioned-scheduling approach for multi-core CPUs, and can also ensure greater predictability, by reducing interference within a partition. Future work will look at relaxing this constraint.

Figure 9.6 illustrates an example accelerator partition. In this example, we consider five accelerator requests $r_i | i \in \{1, 2, ...5\}$ each described by a tuple (G, F), where G is the worst-case execution time of the request on the accelerator and F is the fraction of the accelerator required by the request. These request parameter tuples are described in the following set $\{r_1$ =(2,0.6), r_2 =(1,0.25), r_3 =(1,0.4), r_4 =(5,0.4), r_5 =(3, 0.55)$\}$. We assign requests r_2 and r_4 to partition 1, and requests r_1, r_3 and r_5 to partition 2. Observe that, within a partition, requests execute in serial order.

9.4.1 Partitioning Schedulability Analysis

We now describe the response-time-based schedulability analysis to decide the schedulability of tasksets using partitioned accelerators. In particular, we consider work-conserving fixed-priority scheduling. From a schedulability-analysis perspective, each accelerator partition behaves as a separate non-preemptive shared resource. Addition-

ally, as we assume that only a single task can access a partition, each partition also behaves as a "mutually-exclusive" resource. Therefore, we can directly use the analysis presented in [53] by considering each partition as an individual resource, with access to each partition governed by its own *lock*, arbitrated using the MPCP protocol [105].

As described in Section 9.1.2, the response time of a task is influenced by the time for which a task is (i) preempted by higher-priority tasks, and (ii) the blocking delay faced in accessing a shared resource. However, when we consider accelerator partitions, there is no impact on the preemption caused by higher-priority tasks.

Therefore, building on the MPCP-based blocking analysis for non-concurrent accelerators [53], as described in Section 9.2.1, the blocking faced by tasks is composed of the following components: (i) *direct blocking*, caused by tasks τ_j using a resource requested by τ_i, (ii) *indirect blocking*, caused when tasks τ_k accessing a resource with a higher priority ceiling preempt the execution of τ_j, which is holding a resource that τ_i is waiting for, (iii) *prioritized blocking*, which is incurred when lower-priority tasks τ_l executing their critical sections with priority ceilings preempt the CPU execution of τ_i, and (iv) *concurrency-induced serialization*, incurred when higher-priority tasks τ_h executing concurrently on the shared resource with τ_i block the CPU critical sections of τ_i.

Of the above blocking terms, indirect blocking, prioritized blocking and concurrency-induced serialization, all occur on the CPU. Therefore, their analysis is not impacted by the introduction of accelerator partitions. On the other hand, direct blocking is *only* caused by accelerator requests which execute on the same partition as the blocked request. Therefore, the direct-blocking analysis needs to take into account which partition a request is assigned to, and it can be computed using the request-driven, job-driven or hybrid analyses proposed in [53].

9.4.2 WFD-based Accelerator Partitioning

We now present a heuristic to efficiently partition an accelerator. Unlike the process of assigning tasks to CPU cores, the accelerator-partitioning problem needs to (i) first generate accelerator partitions, which can have different fractional sizes, and (ii) subsequently assign requests to each partition, while ensuring that the fractional requirement $F_{i,j}$ of an accelerator request $\tau_{i,j}$ is less than or equal to the fractional size f_k of the partition v_k. Thus, the second stage of accelerator partitioning is equivalent to a constrained version of the bin-packing problem, which is known to be NP-hard [79]. Therefore, we believe that the accelerator-partitioning problem is also NP-hard, and hence focus on finding an efficient heuristic.

Algorithm 17 presents a worst-fit decreasing (WFD)-based heuristic called WFD-AcceleratorPartition, for assigning accelerator requests to partitions. As mentioned in Section 6.5.1, among the task-partitioning heuristics studied in the literature, the *Worst-Fit Decreasing* (WFD) algorithm is known to typically produce a well-balanced partition [79]. WFD allocates tasks one by one in non-decreasing order of their utilization. Given a task to be allocated, WFD allocates it to the partition with the least utilization. This makes WFD useful in this setting, as it ensures that all accelerator partitions have similar utilization, thus providing a greater chance for the taskset to be schedulable.

Our proposed heuristic first generates an initial set of partitions, each with an equal fractional size, equivalent to the accelerator request with the biggest fractional requirement. Note that the request with the biggest fractional requirement dictates the size of the largest accelerator partition. Subsequently, requests are assigned to the created partitions using the worst-fit decreasing strategy, while taking into account whether a request $\tau_{i,j}$ can be accommodated into a partition v_k (based on its fractional requirement $F_{i,j} \leq f_k$). Each partition is then re-sized based on the request with the biggest fractional requirement $f_k = max_{\tau_{i,j} \in v_k} F_{i,j}$. Any left-over fraction after the previous step, is assigned to a new partition. All these steps are performed iteratively until either (i) no "left-over"

Algorithm 17 WFD-based Accelerator Partitioning

```
1: procedure WFD-AcceleratorPartition(Γ)
2:     ρ = OrderAccRequestsByUtil(Γ)          ▷ order accelerator requests by utilization
3:     F_large = FindLargestAccFraction(ρ)    ▷ largest accelerator fractional requirement
4:     num_partitions = ⌊1/F_large⌋           ▷ initial number of partitions
5:     Y = InitialPartitions(num_partitions)  ▷ initialize the partitions of the same size
6:     left_over = 1                          ▷ initialize the left over fraction
7:     while left_over > 0 do
8:         /* Use WFD to assign requests to partitions */
9:         status = WorstFitDecreasing(ρ, Y)
10:        /* Check if at least one partition is empty */
11:        if status is False then
12:            break
13:        /* Re-size each partition to the assigned request with the biggest fraction */
14:        left_over = ResizePartitions(ρ, Y)
15:    return
16: procedure WorstFitDecreasing(ρ, Y)
17:    /*ρ = list of accelerator requests */
18:    /*Y = list of partitions */
19:    for req in ρ do
20:        if req is movable then
21:            /* Assign the request to the emptiest partition it fits in */
22:            v. = AssignReqToEmptiestFeasiblePartition(Y, req)
23: procedure ResizePartitions(ρ, Y)
24:    /*ρ = list of accelerator requests */
25:    /*Y = list of partitions */
26:    left_over = 1                          ▷ unused accelerator fraction
27:    /* Resize each partition to the maximum fractional requirement */
28:    for v in Y do
29:        /* Get the request req with the maximum fraction in the partition v*/
30:        max_fraction, req = FindMaxFractionPartition(v)
31:        resize(v, max_fraction)            ▷ resize the partition
32:        /* make req immovable, and all others in partition movable */
33:        left_over = left_over - max_fraction
34:        MarkReqFixed(v, req)
35:    CreateNewPartition(Y, left_over)       ▷ create a new partition from the left over
   return left_over
```

fraction is available to create a new partition, or (ii) the worst-fit decreasing step yields one partition which does not contain any requests. Case (i) implies that we cannot create any new partitions, without re-assigning requests, and Case (ii) indicates that the smallest partition cannot accommodate any of the requests in the taskset. In summary,

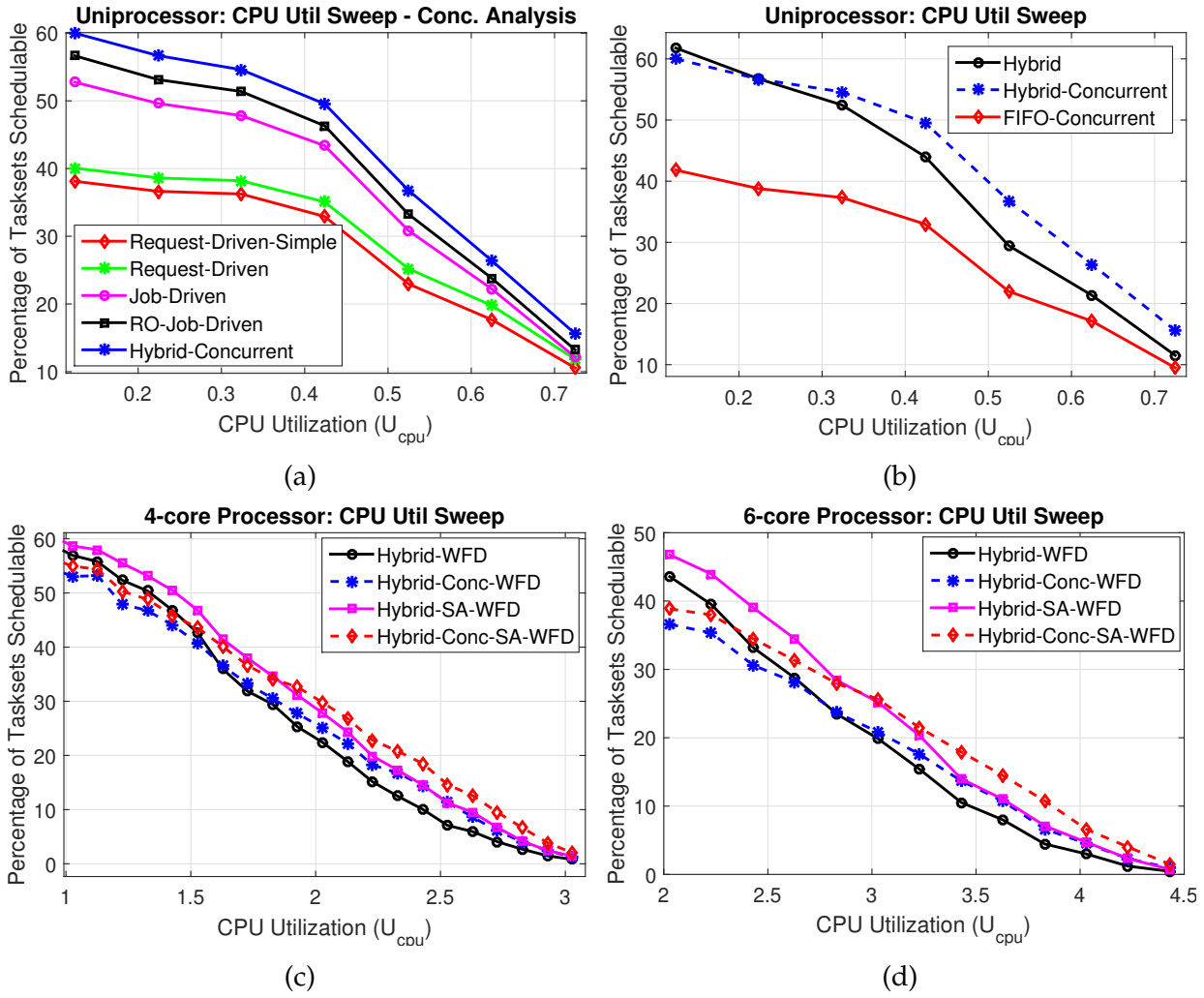


Figure 9.7: Schedulability vs CPU Utilization: (a) unicore: concurrent analyses, (b) unicore, (c) 4-core and, (d) 8-core concurrent vs traditional analysis.

our heuristic iteratively creates new partitions from the left-over fraction, and tries to balance the requests among the existing partitions. While doing so, we also ensure that the request with the biggest fraction in each partition is not moved by WFD in the next iteration. This ensures that the partition sizes do not decrease between iterations.

9.5 Comparative Evaluation

We now present an analytical evaluation of our proposed schedulability-analysis techniques for concurrent hardware accelerators. On the CPU, we assume fully-partitioned

fixed-priority scheduling, with task priorities assigned using the Rate-Monotonic policy [29]. To the best of our knowledge, no other analysis techniques exist for concurrent hardware accelerators. Therefore, we compare against the *traditional* fixed-priority schedulability-analysis introduced in prior work, which models the accelerator as a serialized resource [53].

We compare all techniques on the basis of schedulability. Every data point plotted uses 5000 tasksets, randomly generated using the UUniFast-Discard [175] algorithm, such that no task has a CPU or accelerator utilization greater than 0.4. We consider sporadic tasks with the minimum inter-arrival time randomly assigned to be between 5 and 500 time units.

Unicore Experiments: Figures 9.7a and 9.7b plot the percentage of tasksets schedulable as we vary the CPU utilization keeping the accelerator utilization $U_{gpu} = 0.3$ and the maximum fractional requirement of any accelerator request, $F_{max} = 0.5$. Figure 9.7a compares the schedulability of our proposed concurrent schedulability analyses. As expected, the hybrid analysis yields the best schedulability: 7.8% and 43.5% more tasksets than the request-oriented job-driven and the request-driven analyses respectively. This is because the hybrid analysis computes blocking by taking the minimum of the job-driven and request-driven analyses (Equation 9.4).

Figure 9.7b compares the schedulability of our concurrent hybrid analysis (Hybrid-Concurrent) against the traditional hybrid analysis (Hybrid) [53], which treats the accelerator as a serialized entity. We also compare against the proposed non-work-conserving FIFO analysis. For the most part, the concurrent hybrid analysis yields the best schedulability: 8% and 50.1% more tasksets than the traditional hybrid and FIFO analyses respectively. However, when the CPU utilization is low, the traditional hybrid analysis dominates. This is because our proposed concurrent analysis has extra pessimism. This is due to the lower-priority blocking suffered because of the work-conserving nature of the fixed-priority scheduler, which leads to a few tasksets becoming unschedulable.

Multicore Experiments: Figures 9.7c and 9.7d plot the percentage of tasksets schedu-

lable as we vary the CPU utilization of a 4-core (Figure 9.7c) and 6-core processor (Figure 9.7d), keeping the accelerator utilization $U_{gpu} = 0.3$ and the maximum fractional requirement of any accelerator request $F_{max} = 0.5$. We compare our proposed concurrent hybrid analysis with the traditional serialized hybrid analysis. As we consider fully-partitioned scheduling on the CPU, we also compare two partitioning techniques namely Worst-Fit Decreasing (WFD) and Sync-Aware WFD (SA-WFD) [60]. Note that while allocating tasks to cores the partitioning techniques use schedulability analysis to decide feasibility. Like the unicore case, for the most part, the concurrent hybrid analysis yields better schedulability. Observe that, when the CPU utilization is low, the traditional analysis dominates, but after a certain utilization ($\sim$1.4 and $\sim$2.3 for the 4 and 6 core cases respectively), the trend flips and the concurrent analysis dominates. Among partitioning techniques, Sync-Aware WFD yields better schedulability – 17.8% more tasksets than WFD on the 6-core setting, if we consider our concurrent hybrid analysis. This is because SA-WFD constrains the tasks using the accelerator to a few cores, which restricts the effects of blocking to these cores [60].

Figures 9.8a, 9.8b, 9.8c and 9.8d plot the percentage of tasks schedulable on a 4-core processor, while varying various accelerator-related taskset parameters, and keeping the CPU utilization $U_{cpu} = 1.5$. In all four plots, among partitioning techniques, SA-WFD yields better schedulability than WFD. Therefore, we summarize the differences between the traditional (serialized) and concurrent hybrid analysis using SA-WFD as the partitioning technique. Figure 9.8a varies the accelerator (GPU) utilization, while keeping the maximum fractional requirement of any accelerator request $F_{max} = 0.5$. As the accelerator utilization increases, the concurrent hybrid approach can schedule up to 49.02% more tasksets than the traditional hybrid analysis. Figure 9.8b varies the maximum fractional requirement of any accelerator request F_{max}, while keeping the accelerator utilization $U_{gpu} = 0.5$. As expected, when all requests demand a small fraction of the accelerator, the concurrent approach can schedule upto 2.7x more tasks than the traditional hybrid analysis. Alternatively, when all tasks demand a higher fraction of the accelerator, the

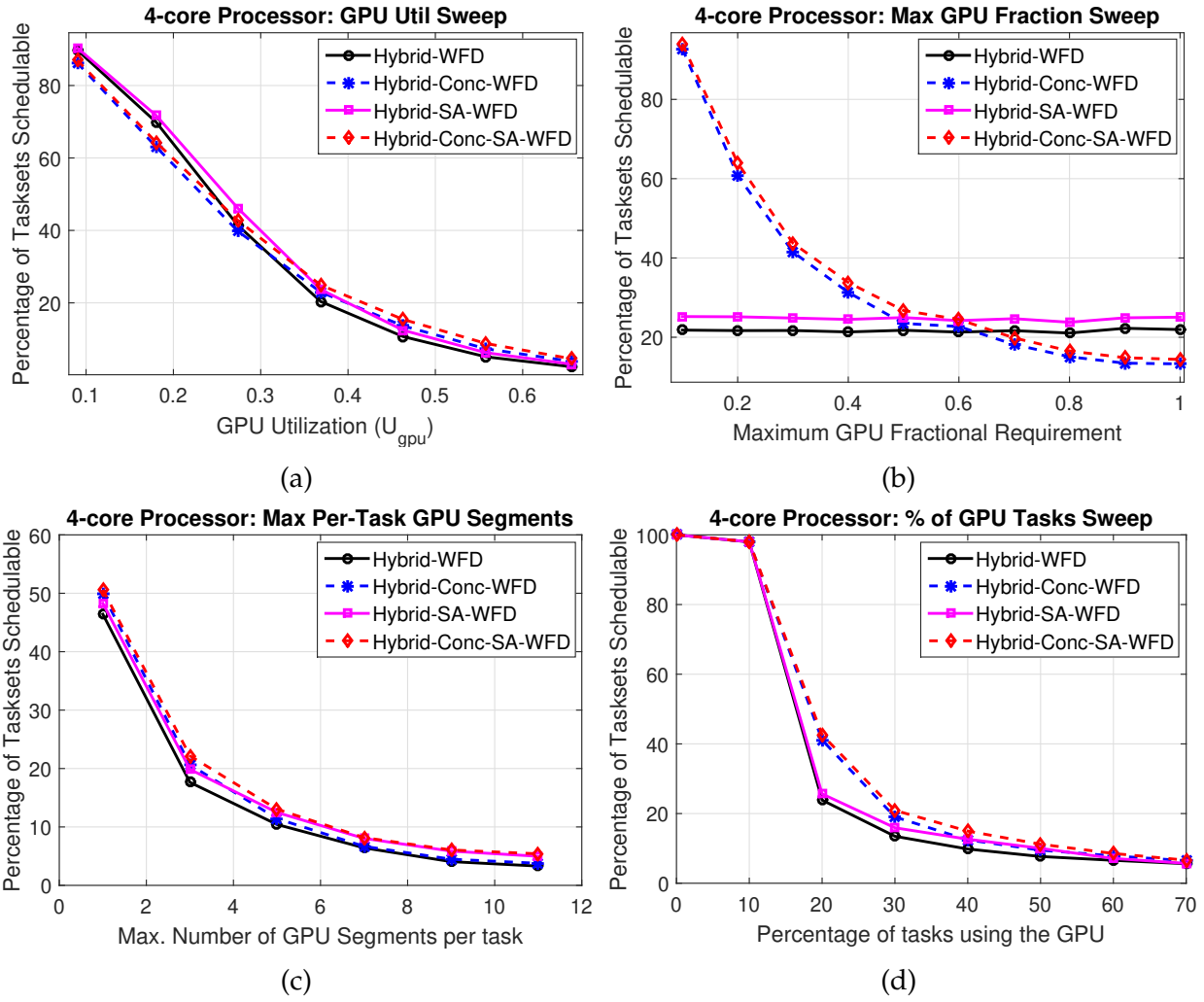
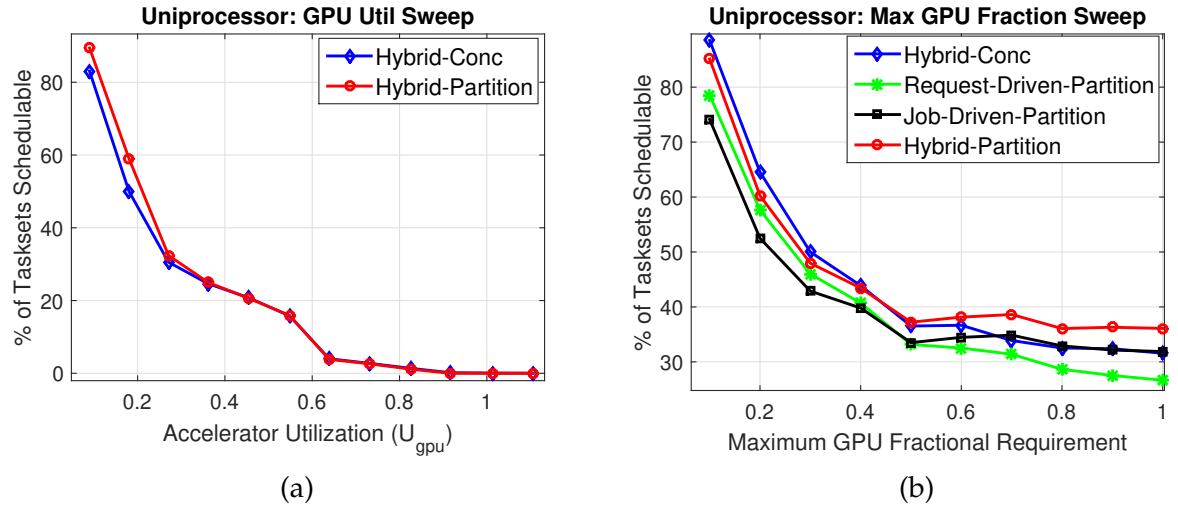


Figure 9.8: Schedulability vs Taskset Accelerator Parameters: (a) accelerator utilization, (b) maximum accelerator fractional requirement of task requests, (c) maximum number of per-task accelerator segments and, (d) percentage of tasks using the accelerator.

traditional hybrid analysis dominates – up to 42.7% greater.

Figure 9.8c varies the maximum number of accelerator requests a job of a task can have, while keeping $U_{gpu} = 0.5$, and $F_{max} = 0.5$. The concurrent hybrid approach schedules 6% more tasksets than the traditional approach. Figure 9.8d varies the percentage of tasks which utilize the accelerator. Again, the concurrent hybrid approach can schedule 10.1% more tasksets than the traditional approach.

Our proposed concurrent analysis techniques on average yield much better schedulability than the traditional non-concurrent analysis. However, there are some scenarios

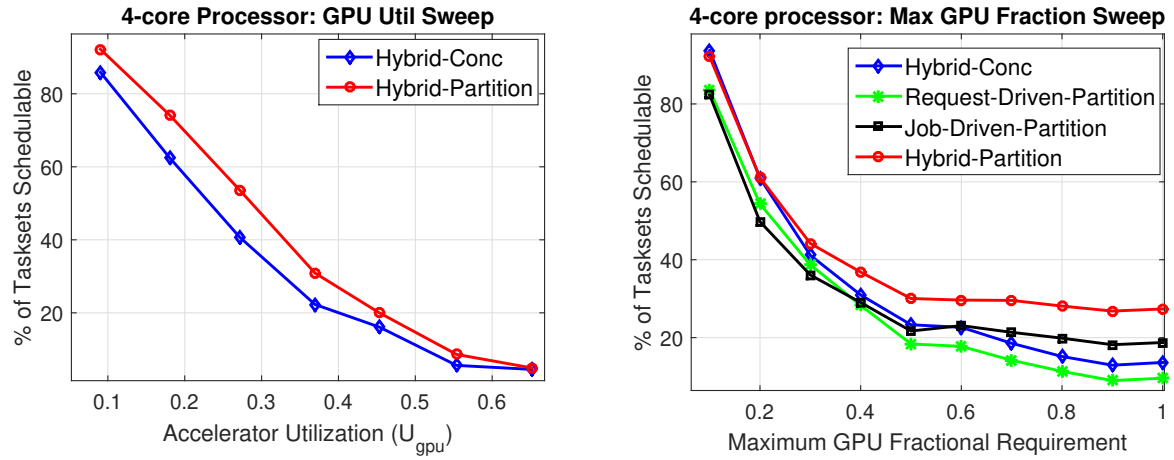


(a) (b)

Figure 9.9: Uniprocessor + Partitioned-Accelerator Schedulability: (a) accelerator utilization, and (b) maximum accelerator fractional requirement of task requests

where the non-concurrent analysis can yield better schedulability, especially when there is lesser concurrency in the system due to tasks with large accelerator fractional requirements ($F_{max} >= 0.5$). This is due to the pessimism our analysis adds for task requests with larger fractions, which, due to the work-conserving nature of the scheduler, can be blocked by lower-priority requests with smaller fractional requirements.

Accelerator-Partitioning Results: We now compare the schedulability of our proposed concurrent-accelerator analysis techniques (global scheduling) against that of our proposed WFD-based accelerator partitioning heuristic. Figures 9.9a and 9.9b plot schedulability for a unicore processor as a function of (a) accelerator utilization (keeping the maximum fractional requirement of any accelerator request, $F_{max} = 0.5$), and (b) the maximum fractional requirement of any accelerator request (keeping the accelerator utilization $U_gpu = 0.4$). As we vary the accelerator utilization (Figure 9.9a), we observe that at lower utilization, the hybrid partitioned analysis (Hybrid-Partition) can schedule up to ~18% more tasksets than the hybrid global scheduling analysis (Hybrid-Conc). Similarly, as we increase the maximum fractional requirement of the accelerator (Figure 9.9b), we observe that the partitioned scheduling approach can schedule up to ~14% more tasksets than the hybrid global scheduling analysis.



(a) (b)

Figure 9.10: Multicore + Partitioned-Accelerator Schedulability: (a) accelerator utilization, and (b) maximum accelerator fractional requirement of task requests

Figures 9.10a and 9.10b plot schedulability for a quad-core processor as a function of (a) accelerator utilization (keeping the maximum fractional requirement of any accelerator request, $F_{max} = 0.5$), and (b) the maximum fractional requirement of any accelerator request (keeping the accelerator utilization $U_gpu = 0.4$). Observe that at lower accelerator utilization (Figure 9.10a), the hybrid partitioned analysis (Hybrid-Partition) can schedule up to ~53% more tasksets than the hybrid global scheduling analysis (Hybrid-Conc). Similarly, like the uniprocessor case, as we increase the maximum fractional requirement of the accelerator (Figure 9.10b), the partitioned-scheduling approach can schedule up to ~107% more tasksets than the hybrid global-scheduling analysis.

In summary, we observe that, as the maximum accelerator fraction of accelerator requests increases, the accelerator partitioning approach yields significantly higher schedulability than the concurrent global scheduling approach. This is because the concurrent global analysis, significantly penalizes higher-priority requests with larger fractional requirement of the accelerator (Lemma 4).

9.6 Summary

Traditional analyses consider an accelerator as a serial resource, which can lead to schedulability pessimism. Therefore, in this chapter, we introduced schedulability-analysis techniques for real-time systems with hardware accelerators supporting concurrent execution. For the concurrent accelerator, we consider non-preemptive *global* scheduling, and propose analysis techniques for both work-conserving fixed-priority scheduling and non work-conserving FIFO scheduling.

We first characterized and defined the scheduling problem and proved multiple properties associated with scheduling requests concurrently on a hardware accelerator. In particular, we proposed the Multi-Access MPCP protocol to govern concurrent access to the accelerator, while allowing concurrency-induced serialization on the CPU to proceed in task-priority order. We also formulated the scheduling problem as the maximum-resource bin-packing problem, and proposed the *wavefront* and *liquefaction* techniques which provide an upper bound on the blocking faced by a task's accelerator request.

Subsequently, for work-conserving fixed-priority scheduling, we generalized our proposed methodologies to derive a schedulability-analysis framework based on the request-driven, job-driven and hybrid analyses [53] [54] [55]. We also introduced an analysis for non work-conserving FIFO scheduling. Evaluations show that when there is significant opportunity for concurrency, our analysis techniques yield increased schedulability over non-concurrent analyses.

However, due to the inherent pessimism of the analysis, there is scope for improved schedulability. Therefore, we utilize the ability to partition compute and memory resources in modern GPUs and propose partitioned-scheduling techniques which both increase schedulability and decrease interference. We therefore recommend the use of partitioned scheduling.

Chapter 10

Conclusions and Future Work

The primary contribution of this dissertation is the development of novel distributed software abstractions and frameworks, which in conjunction with node-level analytical real-time scheduling techniques, enable resource-efficient and time-aware geo-distributed coordination in cyber-physical systems. In particular, we highlighted the necessity of a shared-notion of time to enable coordination at the *distributed* scope, along with the importance of simultaneously scheduling multiple application components at the scope of each *node*, such that all deadlines are met, while ensuring that the resource/-physical constraints of the system are satisfied.

To support our thesis statement, we introduced multiple components which can be used stand-alone or be combined together to enable both time-aware and energy-efficient cyber-physical systems. Our research contributions are summarized as follows:

- **Timelines, Quality of Time (QoT) and the QoT Stack [17] [4]:** In Chapter 3 we introduced the concept of *Quality of Time* (QoT) as the "end-to-end uncertainty in the notion of time delivered to an application by the system". Building on the notion of QoT, we also introduced the QoT Stack, centered around a shared *virtualized* notion of time, based on the *timeline* abstraction, which allows applications to specify their timing requirements, while delivering the required QoT and exposing

timing uncertainty to applications. Lastly, we also argued that the knowledge of QoT enables applications to adapt and be fault-tolerant, while allowing the system to manage resources efficiently.

- **Bringing QoT to Virtual Machines [24]:** Most public clouds and edge platforms provide multi-tenancy using *virtualized* units of computing. Therefore, in Chapter 4 we introduced the notion of QoT to virtual machines. The use of virtualization presents a challenge in terms of observing and guaranteeing the QoT delivered to an application. To meet these challenges, we presented the QuartzV extension to the QoT Stack, to make virtual machines QoT-aware. QuartzV utilizes *para-virtual* clocks, which our experiments indicate are key for delivering near-native levels of *timing* performance in virtual machines.

- **Time-as-a-Service for Geo-distributed Coordination [18] [28]:** In Chapter 5 we focused on enabling fault-tolerant *time-based* coordinated applications running on multi-tenant geo-scale infrastructure. To enable such applications, we introduced the Quartz framework, which provides *Time-as-a-Service*. We defined *Time-as-a-Service* (TaaS) as "the ability to provide an application-specific clock, which tracks a time reference, such that the timing uncertainty does not exceed application-specified requirements." Quartz allows geo-distributed application components to each specify its timing requirements, while it *autonomously* orchestrates the underlying infrastructure to meet them. Quartz is designed for containerized applications, features a distributed architecture and is implemented using containerized micro-services. Our experimental evaluations indicate that Quartz can be deployed on real-world embedded, edge and cloud platforms and can provide Time-as-a-Service in a geo-distributed setting.

- **Energy-Saving Multi-core Real-Time Sleep Schedulers [58]:** Modern processors provide sleep states which minimize leakage power by gating portions of the processor and/or the system clock. In Chapter 6, we presented partitioned fixed-

priority scheduling solutions for utilizing the processor *deep*-sleep state to efficiently schedule sporadic real-time tasks, and maximize energy savings. The techniques presented rely on our proposed family of *Energy-Saving* schedulers namely, an enhanced version of *Energy-Saving Rate-Harmonized Scheduling* (ES-RHS) [42], and our newly proposed *Energy-Saving Rate-Monotonic Scheduling* (ES-RMS) policy to maximize the time the processor spends in the lowest-power *deep-sleep* state. We also illustrated the benefits of using ES-RMS over ES-RHS for processors which only allow all cores to transition into the deep-sleep state together. For processors which allow cores to individually transition into deep sleep, we prove that, while utilizing ES-RHS on each core, any feasible partition can optimally utilize all of the processor's idle durations to put it into deep sleep.

- **Thermal Implications of Energy-Saving Schedulers [59]:** In Chapter 7, we explore the relationship between energy savings and system temperature in the context of our previously-proposed fixed-priority *energy-saving* schedulers, which utilize a processor's *deep-sleep* state to save energy. We derive insights from a well-known thermal model, and identified *proactive* design choices which are *independent* of system constants and can be used to reduce processor temperature. Based on these insights, we proposed the SysSleep and ThermoSleep algorithms, which enable both an energy-efficient and thermally-effective sleep schedule. Our observations and experiments indicate that, while energy savings are key to lower temperatures, not all energy-efficient solutions yield low temperatures.

- **Energy-Saving Scheduling for Real-Time Systems with Hardware Accelerators [60]:** In many modern cyber-physical systems, tasks execute using a combination of CPU and accelerator resources such as GP-GPUs. Hence, in Chapter 8 we focus on reducing the energy consumption of systems using hardware accelerators. To reduce energy consumption, commercially-available accelerators such as GP-GPUs and DSPs are equipped with interfaces to scale their operating voltage and fre-

quency. Hence, we proposed the CycleTandem static frequency-scaling technique to *co-optimize* the operating frequencies of both the CPU and the hardware accelerator. Based on practical considerations of real-world platforms, we also considered various energy-management scenarios where the accelerator or CPU frequencies may or may not be adjustable, and proposed the CycleSolo family of algorithms for such contexts. To evaluate our proposed techniques, we performed both analytical evaluations and real-world experiments on the NVIDIA TX2 platform, on which we observed up to 44.29% lower energy consumption as compared to the case without energy management.

- **Co-Scheduling Real-Time Workloads on Concurrent Hardware Accelerators:** Modern accelerators often support concurrent execution, and allow requests belonging to different tasks to be *co-scheduled* and execute in parallel. However, existing fixed-priority real-time scheduling analyses assume that tasks can access the accelerator only one at a time. Therefore, in Chapter 9, we propose schedulability-analysis techniques for real-time tasksets utilizing hardware accelerators which support concurrent execution. We first considered *global* scheduling, where the accelerator is treated as a single resource, and focused on work-conserving fixed-priority scheduling and non-work-conserving FIFO scheduling. Subsequently, we also considered *partitioned*-scheduling techniques, where an accelerator can be partitioned into discrete compute units, and the requests in the taskset can be allocated to these partitions. Our experimental evaluations suggest that our proposed analysis methodologies can yield improved schedulability, up to ~2x more tasksets, over traditional non-concurrent analysis techniques.

10.1 Future Directions

The future holds promise for cyber-physical system with possibly even inter-planetary-scale coordination. Consider the recently launched Breakthrough Starshot Initiative [187]. The initiative proposes to send laser-propelled nano-spacecraft to Alpha Centauri. While there are a number of engineering challenges still to be solved, one can envision a global network of laser arrays being used to send such nano-spacecraft into deep space. These laser arrays will typically be *fired* for a few minutes to accelerate the space-craft to about 20% of the speed of light [187]. To propel fleets of nano-spacecrafts, it will be essential to precisely coordinate the direction and intensity of these geographically-distributed laser arrays, while taking into account the effects of the earth's rotation and atmospheric interference. One can also envision an inter-planetary network of such lasers.

The inherent complexity and geo-distributed nature of such cyber-physical applications along with the heterogeneity of the infrastructure makes their development, deployment and management a challenging proposition. This has often resulted in the development of application-specific siloed solutions, which are often over-engineered. Therefore, what is required is a distributed framework which solves a key set of challenges common to a range of these systems. This dissertation focused on two challenges, namely *time* and *energy*. However, given the spatio-temporal nature of CPS, we posit that in the future both *time* and *location* need to be exposed as first-class entities to cyber-physical applications, while meeting the resource constraints of the underlying system. Therefore, we advocate for future research which focuses on:

- a distributed cyber-physical OS to enable spatio-temporal coordination in CPS, and
- quality metrics which expose both time and location as first-class primitives to cyber-physical applications.

Therefore, like our proposed *Quality of Time* (QoT) [4] metric, one can also envision

a *Quality of Location* (QoL) [18] metric, which quantifies "the uncertainty radius in a location estimate, with respect to a reference".

While time, location and resource constraints like energy and maximum temperature are paramount in the design of cyber-physical systems, we also believe that the following are some important directions of future research:

- **Security:** The real-world implications of cyber-physical systems make security an important concern. Unlike software services, which are hosted in a secure data center, cyber-physical systems may have physical nodes deployed in public or semi-private spaces. Hence, malicious nodes need to be detected and isolated without violating safety constraints. From a spatio-temporal coordination aspect, time or location spoofing can also adversely impact coordination and result in unsafe decisions. Therefore, for a cyber-physical OS, essential security features include (i) access control to resources, (ii) isolation between applications and users, (iii) anomaly detection and (iv) a verifiable implementation.

- **Privacy:** From a privacy standpoint, significant amounts of personal data may be used by cyber-physical systems for distributed decision making. It is important that all personally-identifiable data be sufficiently anonymized [13] [188] to reduce the risk of exposure in case of data breaches.

- **Fault Tolerance and Reliability:** Most cyber-physical applications are safety-critical [11], making fault tolerance necessary [189]. In the software services domain, fault tolerance is concerned with reducing down time, and preventing information loss. Hence, most services are replicated across different fault-tolerance domains. Most CPS also utilize replication techniques to ensure fault tolerance. However, CPS interact with the real world, where the safety of humans and infrastructure is critical. Therefore, CPS may also rely on *analytical redundancy* [126], involving graceful-degradation modes. When multiple components fail, the system must be able to gracefully degrade and stop without causing any harm [19].

- **Time and Software Systems:** While a shared notion of time is key for cyber-physical coordination, the core-concept of Time-as-a-Service is also useful for distributed software applications [6] [16]. We believe that the ability to request and observe application-specific QoT can be used to relax many of the stringent asynchronous assumptions associated with distributed systems.

Bibliography

[1] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, Oct 1991. xi, 6, 21, 31, 32, 36, 37, 57, 94, 102, 107, 109, 120

[2] J. Eidson and K. Lee. IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems. In *2nd ISA/IEEE Sensors for Industry Conference*, pages 98–105. IEEE, 2002. xi, 6, 21, 32, 37, 61, 94, 97, 102, 103, 104, 107, 110, 116

[3] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 81–94, 2018. xi, 21, 22, 94, 102, 104, 107, 112

[4] F. Anwar, S. D'souza, A. Symington, A. Dongare, R. Rajkumar, A. Rowe, and M. Srivastava. Timeline: An operating system abstraction for time-aware applications. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 191–202. IEEE. xii, 6, 7, 13, 31, 32, 38, 39, 41, 42, 44, 60, 64, 65, 66, 70, 87, 91, 96, 252, 256

[5] G. Regula and B. Lantos. Formation control of a large group of uavs with safe path planning and obstacle avoidance. In *2014 European Control Conference (ECC)*, pages 1522–1529, June 2014. 1, 9

[6] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, 2012. USENIX Association. 1, 24, 33, 60, 90, 258

[7] S. Natarajan and A. Ganz. Surgnet: an integrated surgical data transmission system for telesurgery. *International journal of telemedicine and applications*, 2009:3, 2009. 1

[8] J. J. Enright and P. R. Wurman. Optimization and coordinated autonomy in mobile fulfillment systems. In *Proceedings of the 9th AAAI Conference on Automated Action Planning for Autonomous Mobile Robots*, AAAIWS'11-09, pages 33–38. AAAI Press, 2011. 1, 4

[9] M. Buevich, X. Zhang, O. Shih, D. Schnitzer, T. Escalada, A. Jacquiau-Chamski, J. Thacker, and A. Rowe. Microgrid losses: When the whole is greater than the sum of its parts. In *2016 ACM/IEEE 7th International Conference on Cyber-Physical Systems (ICCPS)*, pages 1–10, April 2016. 1

[10] SAE J2735 standard. https://ntl.bts.gov/lib/51000/51100/51167/DE156ECC.pdf. 1

[11] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. Cyber-physical systems: The next computing revolution. In *Design Automation Conference*, pages 731–736, June 2010. 2, 257

[12] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA, 2012. ACM. 2, 123

[13] P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, and M. Satyanarayanan. Scalable crowd-sourcing of video from mobile devices. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 139–152, New York, NY, USA, 2013. ACM. 2, 257

[14] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, Oct 2009. 2

[15] C. Hung, G. Ananthanarayanan, P. Bodik, L. Golubchik, M. Yu, P. Bahl, and M. Philipose. VideoEdge: Processing Camera Streams using Hierarchical Clusters. pages 115–131, 10 2018. 2

[16] B. Liskov. Practical Uses of Synchronized Clocks in Distributed Systems. *Distrib. Comput.*, 6(4):211–219, July 1993. 2, 20, 33, 35, 60, 90, 258

[17] S. D'souza and R. Rajkumar. Time-based Coordination in Geo-Distributed Cyber-Physical Systems. In *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17)*, Santa Clara, CA, July 2017. USENIX Association. 2, 6, 13, 55, 60, 64, 90, 252

[18] S. D'souza and R. Rajkumar. A Cyber-Physical OS for Enabling Spatio-Temporal Coordination at Geo-distributed Scale. In *Workshop on Next Generation OS for Cyber-Physical Systems (NGOSCPS)*, 4 2019. 3, 14, 89, 253, 257

[19] J. Wei, J. M. Snider, J. Kim, J. M. Dolan, R. Rajkumar, and B. Litkouhi. Towards a viable autonomous driving research platform. In *2013 IEEE Intelligent Vehicles Symposium (IV)*, pages 763–770. IEEE, 2013. 4, 9, 257

[20] Microsoft Hololens. https://www.microsoft.com/en-us/hololens. 4, 177

[21] NVIDIA Xavier. https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit. 4, 12, 214

[22] NVIDIA Jetson TX2. https://developer.nvidia.com/embedded/jetson-tx2. 4, 29, 205, 210, 211

[23] Raspberry Pi 3. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. 4

[24] S. D'souza and R. Rajkumar. QuartzV: Bringing Quality of Time to Virtual Machines. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 49–61. IEEE, 2018. 6, 7, 14, 21, 253

[25] T. Broomhead, L. Cremean, J. Ridoux, and D. Veitch. Virtualize Everything but Time. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 451–464, Berkeley, CA, USA, 2010. USENIX Association. 6, 21, 31, 57, 86

[26] J. E. Smith and R. Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, May 2005. 7, 55

[27] Docker Inc. Docker: Enterprise Container Platform. https://www.docker.com/. 7, 55, 56, 87, 94

[28] S. D'souza, H. Koehler, A. Joshi, S. Vaghani, and R. Rajkumar. Quartz: Time-as-a-Service for Coordination in Geo-Distributed Systems. In *2019 IEEE/ACM Symposium on Edge Computing, SEC 2019*, 2019. 7, 14, 253

[29] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973. 8, 125, 129, 130, 205, 228, 246

[30] WindRiver Inc. VXWORKS. https://www.windriver.com/products/vxworks/. 8

[31] Blackberry QNX. http://blackberry.qnx.com. 8

[32] Linux. https://www.linux.org/. 8

[33] The FreeRTOS Kernel. https://www.freertos.org/. 8

[34] S. K Dhall and C. L. Liu. On a real-time scheduling problem. *Operations research*, 26(1):127–140, 1978. 9

[35] NVIDIA Multi-Process Service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf. 9, 30, 240

[36] NVIDIA Tesla V100. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. 9, 12, 18, 214

[37] S. Jain, I. Baek, S. Wang, and R. Rajkumar. Fractional GPUs: Software-based Compute and Memory Bandwidth Reservation for GPUs. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 29–41. IEEE, 2019. 9, 13, 18, 30, 214, 240

[38] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The impact of technology scaling on lifetime reliability. In *International Conference on Dependable Systems and Networks*, pages 177–186, 2004. 9

[39] R. Viswanath, V. Wakharkar, A. Watwe, and V. Lebonheur. Thermal performance challenges from silicon to systems. *Intel Corp. Manufacturing Group*, 2000. 9

[40] T. Kuroda. CMOS design challenges to power wall. In *International Microprocesses and Nanotechnology Conference*, pages 6–7, 2001. 10

[41] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *IEEE Annual Foundations of Computer Science*, pages 374–382, 1995. 10, 24, 123

[42] A. Rowe, K. Lakshmanan, H. Zhu, and R. Rajkumar. Rate-harmonized scheduling for saving energy. In *IEEE Real-Time Systems Symposium*, pages 113–122, 2008. 10, 15, 24, 26, 27, 123, 124, 125, 126, 129, 135, 147, 151, 153, 254

[43] K. Agarwal, K. Nowka, H. Deogun, and D. Sylvester. Power gating with multiple sleep modes. In *International Symposium on Quality Electronic Design*, pages 633–637, 2006. 10, 24, 123

[44] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, 2003. 10, 24, 149

[45] E. Le Sueur and G. Heiser. Slow Down or Sleep, That is the Question. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 16–16, Berkeley, CA, USA, 2011. USENIX Association. 10, 25

[46] S. Mittal and J. S. Vetter. A survey of methods for analyzing and improving gpu energy efficiency. *ACM Computing Surveys (CSUR)*, 47(2):19, 2015. 10, 177

[47] NVIDIA GPU Performance State Interface. https://docs.nvidia.com/gameworks/content/gameworkslibrary/coresdk/nvapi/group__gpupstate.html. 11, 177

[48] ATI Dynamic Power Management. https://wiki.archlinux.org/index.php/ATI#Dynamic_power_management. 11, 177

[49] P. Glaskowsky. NVIDIA's Fermi: The First Complete GPU Computing Architecture. NVIDIAWhitePaper,2009. 12, 214

[50] Pascal GPU Architecture. https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture/. 12, 214

[51] H. Kim, P. Patel, S. Wang, and R. Rajkumar. A server-based approach for predictable GPU access control. In *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10, Aug 2017. 12, 28, 216

[52] G. A. Elliott, B. C. Ward, and J. H. Anderson. GPUSync: A framework for real-time GPU management. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 33–44. IEEE, 2013. 12, 28, 178, 179, 215, 216

[53] P. Patel, I. Baek, H. Kim, and R. Rajkumar. Analytical enhancements and practical insights for mpcp with self-suspensions. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 177–189, April 2018. 12, 28, 178, 179, 180, 181, 182, 185, 215, 216, 217, 218, 220, 232, 233, 236, 237, 238, 239, 242, 246, 251

[54] G. A. Elliott and J. H. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems*, 48(1):34–74, 2012. 12, 28, 180, 218, 251

[55] K. Lakshmanan, D. Niz, and R. Rajkumar. Coordinated Task Scheduling, Allocation and Synchronization on Multiprocessors. In *2009 30th IEEE Real-Time Systems Symposium*, pages 469–478, Dec 2009. 12, 28, 180, 181, 182, 205, 218, 232, 251

[56] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. In *2011 USENIX Annual Technical Conference (USENIX ATC'11)*, page 17, 2011. 12, 214

[57] Kernel-based Virtual Machine. http://www.linux-kvm.org/. 14, 55, 56, 57, 58, 59, 76

[58] S. D'souza, A. Bhat, and R. Rajkumar. Sleep Scheduling for Energy-Savings in Multi-core Processors. In *Euromicro Conference on Real-Time Systems*, pages 226–236, 2016. 15, 132, 149, 153, 160, 161, 163, 164, 165, 169, 174, 175, 176, 253

[59] S. D'souza and R. Rajkumar. Thermal implications of energy-saving schedulers. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017. 16, 254

[60] S. D'souza and R. Rajkumar. CycleTandem: Energy-Saving Scheduling for Real-Time Systems with Hardware Accelerators. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 94–106, Dec 2018. 17, 247, 254

[61] J. Serrano, M. Lipinski, T. Wlostowski, E. Gousiou, E. van der Bij, M. Cattin, and G. Daniluk. The white rabbit project. 2013. 21

[62] K. Lee, H. Wang, V. Shrivastav, and H. Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 454–467. ACM, 2016. 21

[63] N. Xu, X. Zhang, Q. Wang, J. Liang, G. Pan, and M. Zhang. An Improved Flooding Time Synchronization Protocol for Industrial Wireless Networks. In *2009 International Conference on Embedded Software and Systems*, pages 524–529, May 2009. 22

[64] S. Ganeriwal, R. Kumar, and M. Srivastava. Timing-sync protocol for sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 138–149. ACM, 2003. 22

[65] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. *SIGOPS Oper. Syst. Rev.*, 36(SI):147–163, December 2002. 22

[66] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh. Efficient network flooding and time synchronization with glossy. In *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 73–84, April 2011. 22

[67] C. Lenzen, P. Sommer, and R. Wattenhofer. PulseSync: An Efficient and Scalable Clock Synchronization Protocol. *IEEE/ACM Transactions on Networking*, 23(3):717–727, June 2015. 22

[68] B. R. Hamilton, X. Ma, Q. Zhao, and J. Xu. ACES: Adaptive Clock Estimation and Synchronization Using Kalman Filtering. In *MobiCom*, 2008. 22

[69] X. Xu. A New Time Synchronization Method for Reducing Quantization Error Accumulation Over Real-Time Networks: Theory and Experiments Evaluation of kalman filtering for network time keeping. In *IEEE Trans. on Industrial Informatics*, 2013. 22, 23

[70] C. Seong, S. Lee, and W. Choi. A new network synchronizer using phase adjustment and feed-forward filtering based on low-cost crystal oscillators. In *IEEE TRANSACTIONS ON INSTRUMENTATION AND MEASUREMENT, VOL. 59, NO. 7*, 2010. 22

[71] P. Derler, T. H. Feng, E. A. Lee, S. Matic, H. D. Patel, Y. Zheo, and J. Zou. PTIDES: A programming model for distributed real-time embedded systems. Technical report, DTIC Document, 2008. 23

[72] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003. 23

[73] J. Zou, S. Matic, and E. A. Lee. PtidyOS: A lightweight microkernel for PTIDES real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*. IEEE, 2012. 23

[74] U. Ramachandran, R. S. Nikhil, J. M. Rehg, Y. Angelov, A. Paul, S. Adhikari, K. M. Mackenzie, N. Harel, and K. Knobe. Stampede: A cluster programming middleware for interactive stream-oriented applications. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1140–1154, 2003. 23

[75] D. Hilley and U. Ramachandran. Stampede RT: Programming Abstractions for Live Streaming Applications. In *27th International Conference on Distributed Computing Systems (ICDCS '07)*, pages 65–65, June 2007. 24

[76] D. Hilley and U. Ramachandran. Persistent Temporal Streams. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 17:1–17:20, Berlin, Heidelberg, 2009. Springer-Verlag. 24

[77] POSIX clock. http://pubs.opengroup.org/onlinepubs/009695399/functions/clock.html, IEEE Standard 1003.1, 2004. 24, 37

[78] S. Saewong and R. Rajkumar. Practical voltage-scaling for fixed-priority rt-systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 106–115, 2003. 25, 153, 176, 183, 185

[79] A. Kandhalu, J. Kim, K. Lakshmanan, and R. Rajkumar. Energy-aware partitioned fixed-priority scheduling for chip multi-processors. In *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, volume 1, pages 93–102, Aug 2011. 25, 134, 136, 137, 204, 243

[80] H. Aydin and Qi Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 9 pp.–, April 2003. 25

[81] T. A. AlEnawy and H. Aydin. Energy-aware task allocation for rate monotonic scheduling. In *11th IEEE Real Time and Embedded Technology and Applications Symposium*, pages 213–223, March 2005. 25

[82] J. Chen, C. Yang, H. Lu, and T. Kuo. Approximation algorithms for multiprocessor energy-efficient scheduling of periodic real-time tasks with uncertain task execution time. In *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 13–23, April 2008. 25

[83] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, Dec 2003. 25, 182, 204

[84] J. J. Chen and T. W. Kuo. Procrastination for leakage-aware rate-monotonic scheduling on a dynamic voltage scaling processor. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems*, LCTES '06, pages 153–162, New York, NY, USA, 2006. ACM. 25

[85] S. Irani, S. Shukla, and R. Gupta. Algorithms for Power Savings. *ACM Trans. Algorithms*, 3(4), November 2007. 25

[86] R. Jejurikar and R. Gupta. Procrastination Scheduling in Fixed Priority Real-time Systems. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '04, pages 57–66, New York, NY, USA, 2004. ACM. 25

[87] L. Niu and G. Quan. Reducing both dynamic and leakage energy consumption for hard real-time systems. In *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '04, pages 140–148, New York, NY, USA, 2004. ACM. 25

[88] J. J. Chen and T. W. Kuo. Procrastination determination for periodic real-time tasks in leakage-aware dynamic voltage scaling systems. In *2007 IEEE/ACM International Conference on Computer-Aided Design*, pages 289–294, Nov 2007. 25

[89] J. Chang, H. N. Gabow, and S. Khuller. A Model for Minimizing Active Processor Time. In *Proceedings of the 20th Annual European Conference on Algorithms*, ESA'12, pages 289–300, Berlin, Heidelberg, 2012. Springer-Verlag. 26

[90] C. Fu, Y. Zhao, M. Li, and C. J. Xue. Maximizing common idle time on multicore processors with shared memory. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(7):2095–2108, July 2017. 26

[91] S. Oikawa and R. Rajkumar. Portable rk: a portable resource kernel for guaranteed and enforced timing behavior. In *Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium*, pages 111–120, June 1999. 26

[92] Samsung Exynos 5800. www.samsung.com/semiconductor/minisite/Exynos/w/solution/mobile_ap/5420/. 26, 132

[93] Y. Chandarli, N. Fisher, and D. Masson. Response time analysis for thermal-aware real-time systems under fixed-priority scheduling. In *IEEE International Symposium on Real-Time Distributed Computing*, pages 84–93, 2015. 26, 27, 150, 151, 169

[94] Y. Fu, N. Kottenstette, C. Lu, and X. D. Koutsoukos. Feedback thermal control of real-time systems on multicore processors. In *ACM International Conference on Embedded Software*, pages 113–122, 2012. 26, 27

[95] A. K. Coskun, T. S. Rosing, and K. Whisnant. Temperature aware task scheduling in mpsocs. In *Design, Automation Test in Europe Conference Exhibition*, pages 1–6, 2007. 26, 27

[96] B. Yun, K. G. Shin, and S. Wang. Predicting thermal behavior for temperature management in time-critical multicore systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 185–194, 2013. 26, 27

[97] P. M. Hettiarachchi, N. Fisher, M. Ahmed, L. Y. Wang, S. Wang, and W. Shi. The design and analysis of thermal-resilient hard-real-time systems. In *IEEE Real Time and Embedded Technology and Applications Symposium*, pages 67–76, 2012. 26, 27

[98] R. Ahmed, P. Ramanathan, and K. K. Saluja. On thermal utilization of periodic task sets in uni-processor systems. In *International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 267–276, 2013. 26, 27

[99] N. Fisher, J. J. Chen, S. Wang, and L. Thiele. Thermal-aware global real-time scheduling on multicore systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 131–140, 2009. 26, 27, 160, 161

[100] S. Wang, J. J. Chen, Z. Shi, and L. Thiele. Energy-efficient speed scheduling for real-time tasks under thermal constraints. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 201–209, 2009. 26

[101] J. J. Chen, S. Wang, and L. Thiele. Proactive speed scheduling for real-time tasks under thermal constraints. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 141–150, 2009. 26, 27

[102] T. Chantem, R. P. Dick, and X. S. Hu. Temperature-aware scheduling and assignment for hard real-time applications on mpsocs. In *Design, Automation and Test in Europe*, pages 288–293, 2008. 26, 27

[103] M. Ahmed, N. Fisher, S. Wang, and P. Hettiarachchi. Minimizing peak temperature in embedded real-time systems via thermal-aware periodic resources. *Sustainable Computing: Informatics and Systems*, 1(3):226–240, 2011. 26, 27

[104] G. A. Elliott and J. H. Anderson. An optimal k-exclusion real-time locking protocol motivated by multi-gpu systems. *Real-Time Systems*, 49(2):140–170, 2013. 28

[105] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings.,10th International Conference on Distributed Computing Systems*, pages 116–123, May 1990. 28, 179, 219, 220, 242

[106] J. J. Chen, G. Nelissen, W. H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. Audsley, et al. Many suspensions, many problems: A review of self-suspending tasks in real-time systems. *Real-Time Systems*, 55(1):144–207, 2019. 28, 234

[107] K. Bletsas, N. Audsley, W. H. Huang, J. J. Chen, and G. Nelissen. Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions. Technical report, CISTER-Research Centre in Realtime and Embedded Computing Systems, 2015. 28, 180, 185, 217

[108] NVIDIA GeForce GTX-1080. https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-1080/specifications. 29

[109] J. Zhong and B. He. Kernelet: High-Throughput GPU Kernel Executions with Dynamic Slicing and Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1522–1532, June 2014. 29

[110] H. Zhou, S. Bateni, and C. Liu. S3DNN: Supervised Streaming and Scheduling for GPU-Accelerated Real-Time DNN Workloads. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 190–201. IEEE, 2018. 29, 214

[111] N. Otterness, M. Yang, J. Amert, T.and Anderson, and F. D. Smith. Inferring the scheduling policies of an embedded cuda gpu. In *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2017. 29, 214

[112] M. H. Santriaji and H. Hoffmann. Merlot: Architectural support for energy-efficient real-time processing in gpus. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 214–226, April 2018. 30

[113] Beaglebone Black. https://beagleboard.org/black. 33, 41, 46, 71, 73, 107

[114] Quantum SA.45s Chip-Scale Atomic Clock. https://www.microsemi.com/products/timing-synchronization-systems/embedded-timing-solutions/components/sa-45s-chip-scale-atomic-clock. 36

[115] OpenSplice Data-Distribution System for Real-Time System. https://www.adlinktech.com/Products/IoT_solutions/Vortex_DDS/Vortex_OpenSplice?lang=en. 41, 42

[116] Siemens RUGGEDCOM RSG2488. http://w3.siemens.com. 46

[117] Salae Logic Pro 16. http://downloads.saleae.com. 50

[118] H. Kim, S. Wang, and R. Rajkumar. Responsive and enforced interrupt handling for real-time system virtualization. In *2015 IEEE 21st International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 90–99, Aug 2015. 55

[119] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003. 56, 57

[120] The Xen Project. https://www.xenproject.org/. 56, 57, 63

[121] Intel VT. https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html. 56, 77

[122] AMD Virtualization Solutions for Data Centers. http://www.amd.com/en-gb/solutions/servers/virtualization. 56

[123] J. Ridoux and D. Veitch. TSCclock Goes Live a demonstration of a robust, accurate replacement to ntpd. 57

[124] KVM Paravirtual Clock. http://www.linux-kvm.org/page/KVMClock. 59

[125] Intel 64 and IA-32 Architectures Software Developer Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf. 59

[126] J. Gertler. Analytical redundancy methods in fault detection and isolation-survey and synthesis. *IFAC Proceedings Volumes*, 24(6):9–21, 1991. 60, 90, 257

[127] A. Agostini, C. Torras, and F. Wörgötter. Integrating task planning and interactive learning for robots to work in human environments. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011. 61

[128] H. S. Bank, S. D'souza, and A. Rasam. Temporal logic (tl)-based autonomy for smart manufacturing systems. *Procedia Manufacturing*, 26:1221–1229, 2018. 61

[129] T. Schouwenaars, E. Feron, and J. P. How. Safe receding horizon path planning for autonomous vehicles. In *Annual Allerton Conference on Communication Control and Computing*, 2002. 61

[130] Universal Robotics UR-5. https://www.universal-robots.com/. 61, 71

[131] Kuka Robotics. https://www.kuka.com. 61

[132] VirtIO-Serial. https://fedoraproject.org/wiki/Features/VirtioSerial. 66

[133] C. Macdonell. Inter-VM shared memory PCI device. https://lwn.net/Articles/380869/. 67

[134] ROS Indigo Igloo. http://wiki.ros.org/indigo. 71

[135] Gazebo Robot Simulation. http://gazebosim.org/. 71

[136] ROS Control. http://wiki.ros.org/ros_control. 71

[137] Moxa EtherDevice Switch EDS-405A-PTP. https://www.moxa.com/product/EDS-405A-PTP.htm. 72, 82

[138] Intel Gigabit CT Adapter. https://www.intel.com/content/www/us/en/products/network-io/ethernet/gigabit-adapters/ct-desktop.html. 73, 74

[139] Endace DAG 7.5G2. https://www.endace.com/dag-7.5g2-datasheet.pdf. 73

[140] Linux Kernel Packet Timestamping. https://www.kernel.org/doc/Documentation/networking/timestamping.txt. 73

[141] The Linux PTP Project. http://linuxptp.sourceforge.net. 74, 104

[142] Ubuntu stress tool. http://manpages.ubuntu.com/manpages/xenial/man1/stress.1.html. 81, 114, 118

[143] iperf: The TCP, UDP and SCTP network measurement tool. https://iperf.fr/. 82, 115, 118

[144] Apache Zookeeper. https://zookeeper.apache.org/. 96

[145] VDSO - overview of the ELF shared object. http://man7.org/linux/man-pages/man7/vdso.7.html. 98

[146] NATS Pub-Sub. https://nats.io/. 100

[147] PTP hardware-clock infrastructure for Linux. https://www.kernel.org/doc/Documentation/ptp/ptp.txt. 100

[148] Docker: Add host device to container. https://docs.docker.com/engine/reference/commandline/run/add-host-device-to-container---device. 100

[149] Ping. https://linux.die.net/man/8/ping. 101

[150] Chrony NTP. https://chrony.tuxfamily.org/. 102

[151] Simulation of Urban Mobility (SUMO). http://sumo.dlr.de/index.html. 106

[152] MQTT connectivity protocol. http://mqtt.org/. 106

[153] Nutanix Xi IoT Platform. https://www.nutanix.com/products/iot. 107, 119

[154] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015. 107

[155] Intel NUC Kit NUC7i3BNK. https://www.intel.in/content/www/in/en/products/boards-kits/nuc/kits/nuc7i3bnk.html. 107

[156] Kubernetes. https://kubernetes.io/. 108

[157] AM335X CPSW Ethernet Driver Guide. http://processors.wiki.ti.com/index.php/AM335x_CPSW_(Ethernet)_Driver27s_Guide. 108, 111

[158] Intel Ethernet Connection I219-V. https://downloadcenter.intel.com/product/82186/Intel-Ethernet-Connection-I219-V. 109

[159] D. Allan. Clock Characterization Tutorial. https://tf.nist.gov/general/pdf/2082.pdf. 111

[160] Amazon Web Services. https://aws.amazon.com/. 119

[161] Microsoft Azure. https://azure.microsoft.com/. 119

[162] Google Cloud Platform. https://cloud.google.com/. 119

[163] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993. 125

[164] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990. 125

[165] Intel Core2 Duo. download.intel.com/design/processor/datashts/320390.pdf. 132

[166] AMD Opteron Family. support.amd.com/TechDocs/40036.pdf. 132

[167] Intel Core Processor Family (4th Generation). www.intel.com/content/dam/www/public/us/en/documents/datasheets/4th-gen-core-family-desktop-vol-1-datasheet.pdf. 133

[168] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969. 137, 138

[169] R. Zhang, M. R. Stan, and K. Skadron. Hotspot 6.0: Validation, acceleration and extension. *University of Virginia*, CS-2015-04. 151, 169, 173

[170] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium*, pages 166–171, 1989. 153, 158, 185, 228

[171] H. Wei, K. Lin, W. Lu, and W. Shih. Generalized rate monotonic schedulability bounds using relative period ratios. *Information Processing Letters*, 107(5):142 – 148, 2008. 155

[172] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co, 1990. 162

[173] J. Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237 – 250, 1982. 162

[174] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations: Symposium on the Complexity of Computer Computations*, pages 85–103, 1972. 163

[175] P. Emberson, R. Stafford, and R. Davis. Techniques for the synthesis of multiprocessor tasksets. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems*, pages 6–11, 2010. 169, 205, 246

[176] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on Workload Characterization*, pages 3–14, 2001. 173

[177] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An evaluation of high-level mechanistic core models. *ACM Trans. Archit. Code Optim.*, 11(3):28:1–28:25, 2014. 174

[178] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, 2009. 174

[179] Qualcomm Snapdragon. https://www.qualcomm.com/snapdragon. 177

[180] NVIDIA Tegra. http://www.nvidia.com/object/tegra.html. 177

[181] M. Peres. Reverse engineering power management on NVIDIA GPUs-A detailed overview. *Power*, 75(75W):150W. 182

[182] K. S. Lakshmanan. Scheduling and synchronization for multi-core real-time systems. 2011. 204

[183] Jetson TX2 - INA226 (Power Monitor with i2C Interface). https://devtalk.nvidia.com/default/topic/1000830/jetson-tx2/jetson-tx2-ina226-power-monitor-with-i2c-interface-/. 211

[184] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith. GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115. IEEE, 2017. 216

[185] S. Lauzac, R. Melhem, and D. Mosse. Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor. In *Proceeding. 10th EUROMICRO Workshop on Real-Time Systems (Cat. No. 98EX168)*, pages 188–195. IEEE, 1998. 222

[186] J. Boyar, L. Epstein, L. M. Favrholdt, J. S. Kohrt, K. S. Larsen, M. M. Pedersen, and S. Wøhlk. The maximum resource bin packing problem. *Theoretical Computer Science*, 362(1-3):127–139, 2006. 223

[187] Starshot Breakthrough Initiative. https://breakthroughinitiatives.org/Initiative/3. 256

[188] S. D'souza, R. Rajkumar, V. Bahl, L. Ao, and Landon Cox. AMADEUS: Scalable, Privacy-Preserving Live Video Analytics. under review. 257

[189] A. Bhat. *Practical Solutions for Fault-Tolerance in Connected and Autonomous Vehicles (CAVs)*. PhD thesis, Carnegie Mellon University. 257