Power-Performance-Predictability: Managing the Three Cornerstones of Resource Constrained Real-Time System Design

Anway Mukherjee

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Thidapat (Tam) Chantem, Chair
Ryan Gerdes
T. Charles Clancy
Eli Tilevich
Guoqiang Yu

August 28, 2019
Arlington, Virginia

Keywords: Real-Time Systems, Trusted Execution, Voltage-Frequency Scaling, Android

Power-Performance-Predictability: Managing the Three Cornerstones of Resource Constrained Real-Time System Design

Anway Mukherjee

(ABSTRACT)

This dissertation explores several challenges that plague the hardware-software co-design of popular resource constrained real-time embedded systems. We specifically tackle existing real-world problems, and address them through our design solutions which are highly scalable, and have practical feasibility as verified through our solution implementation on real-world hardware.

We address the problem of poor battery life in mobile embedded devices caused due to side-by-side execution of multiple applications in split-screen mode. Existing industry solutions either restricts the number of applications that can run simultaneously, limit their functionality, and/or increase the hardware capacity of the battery associated with the system. We exploit the gap in research on performance and power trade-off in smartphones to propose an integrated energy management solution, that judiciously minimizes the system-wide energy consumption with negligible effect on its quality of service (QoS).

Another important real-world requirement in today's interconnected world is the need for security. In the domain of real-time computing, it is not only necessary to secure the system but also maintain its timeliness. Some example security mechanisms that may be used in a hard real-time system include, but are not limited to, security keys, protection of intel-

lectual property (IP) of firmware and application software, one time password (OTP) for software certification on-the-fly, and authenticated computational off-loading. Existing design solutions require expensive, custom-built hardware with long time-to-market or time-to-deployment cycle. A readily available alternative is the use of trusted execution environment (TEE) on commercial off-the-shelf (COTS) embedded processors. However, utilizing TEE creates multiple challenges from a real-time perspective, which includes additional time overhead resulting in possible deadline misses. Second, trusted execution may adversely affect the deterministic execution of the system, as tasks running inside a TEE may need to communicate with other tasks that are executing on the native real-time operating system. We propose three different solutions to address the need for a new task model that can capture the complex relationship between performance and predictability for real-time tasks that require secure execution inside TEE. We also present novel task assignment and scheduling frameworks for real-time trusted execution on COTS processors to improve task set schedulability. We extensively assess the pros and cons of our proposed approaches in comparison to the state-of-the-art techniques in custom-built real-world hardware for feasibility, and simulated environments to test our solutions' scalability.

Power-Performance-Predictability: Managing the Three Cornerstones of Resource Constrained Real-Time System Design

Anway Mukherjee

(GENERAL AUDIENCE ABSTRACT)

Today's real-world problems demand real-time solutions. These solutions need to be practically feasible, and scale well with increasing end user demands. They also need to maintain a balance between system performance and predictability, while achieving minimum energy consumption. A recent example of technological design problem involves ways to improve the battery lifetime of mobile embedded devices, for example, smartphones, while still achieving the required performance objectives. For instance, smartphones that run Android OS has the capability to run multiple applications concurrently using a newly introduced split-screen mode of execution, where applications can run side-by-side at the same time on screen while using the same shared resources (e.g., CPU, memory bandwidth, peripheral devices etc.). While this can improve the overall performance of the system, it can also lead to increased energy consumption, thereby directly affecting the battery life.

Another technological design problem involves ways to protect confidential proprietary information from being siphoned out of devices by external attackers. Let us consider a surveillance unmanned aerial vehicle (UAV) as an example. The UAV must perform sensitive tasks, such as obtaining coordinates of interest for surveillance, within a given time duration, also known as task deadline. However, an attacker may learn how the UAV communicates with ground control, and take control of the UAV, along with the sensitive information it carries.

Therefore, it is crucial to protect such sensitive information from access by an unauthorized party, while maintaining the system's task deadlines.

In this dissertation, we explore these two real-world design problems in depth, observe the challenges associated with them, while presenting several solutions to tackle the issues. We extensively assess the pros and cons of our proposed approaches in comparison to the state-of-the-art techniques in custom-built real-world hardware, and simulated environments to test our solutions' scalability.

Dedication

Safar·nāma | Safar·nāmé | Safarnameh

(To the journey within and beyond)

Acknowledgments

I offer my sincere salutations to my adviser, Dr. Thidapat (Tam) Chantem, the northern star to all my academic pursuits. You took me under your tutelage during the most formative years of my academic life, and nurtured me with a healthy balance of indomitable encouragement and unequivocal objectivity. I am grateful to you for being my overarching support system, especially when the research community often griped with the fruits of my labor. I eagerly look forward to continuing this relationship in future.

To Dr. Nathan Fisher, thank you for being patient with my limited awareness of theoretical knowledge, for which I am forever grateful.

I would also like to thank my committee for believing in my abilities. To Dr. Ryan Gerdes, thank you for the invaluable suggestions that have greatly improved my research endeavors. Special thanks to Dr. Charles Clancy for introducing me to a whole new world of cutting edge research. To Dr. Eli Tilevich, thank you for the incessant incubation of ideas that have greatly buoyed me through my academic sustenance. To Dr. Guoqiang Yu, thank you for always finding time to support some of my most outlandish incongruities.

To Dr. Koushik Chakraborty and Dr. Sanghamitra Roy, thank you for fortifying the fun-

damentals of my academic background at Utah State University. To my mentors Jun Wang and Handong Ye at Huawei Technologies Ltd., thank you for giving me the opportunity for a hands-on experience to deal with real-world industrial problems. Big thanks to all my colleagues whom I have worked with; To Srinath Arunachalam and Jesse Patterson at Utah State, thank you to both of you for taking the time off your busy schedules to help me bounce off research ideas. Special thanks to Tanmaya Mishra at Virginia Tech for his invaluable technical support in my research endeavors, and to being a constant critique of my work. Heartfelt thanks to my extended academic family of RTX Lab at Virginia Tech, particularly, Pratham, Leila, Mahsa, Seth and Gokcen. To my Utah family, thank you for making my stay in Utah memorable.

To Harshita, Nicole, Courtney, Kim, Debapriya, Sreoshi, Diksha, Rumela and Julia, thank you for helping me in this academic sojourn, to all the lovely memories that will always remain etched in my heart, and to the stories that underline my growth as an individual.

Finally, to my parents, baba and maa, I am indebted for all the sacrifices they had to make for my pursuit of knowledge. Thank you for always allowing me to do what I wanted, and never doubting my life choices, even if it meant that you had to shoulder the brunt of hardship for me. To my sister Ananya, thank you for acting way out of your age, and showering me with never-ending love and affection. You are, and will always be my beating heart.

And, to Calcutta, my muse, my eternal flame, milady.

Contents

List of Figures xv

List of Tables xx

1 Introduction 1

1.1 Motivation 2

1.1.1 Power Vs Performance as Design Challenge 2

1.1.2 Performance Vs Predictability as Design Challenge 2

1.2 Major Contributions and Organization 4

2 Power and Performance Issues on Resource Constrained Mobile Embedded Systems 7

2.1 Background 8

2.2 Existing Solutions 9

2.3 Challenges 10

3 Energy Management of Multiple Side-by-Side Applications in Smartphones 12

3.1 Contribution . . . 14
3.2 Proposed Framework . . . 17
3.3 Offline Power-Performance Profiling . . . 21
3.4 Offline Phase Analysis . . . 23
3.4.1 L1 Cache Monitor . . . 25
3.4.2 Last Level Cache Monitor . . . 25
3.4.3 Phase Detector . . . 28
3.4.4 Temporal Phase Profiling . . . 30
3.5 Online Resource Management . . . 32
3.5.1 Constructing Compatible Application Tuples . . . 33
3.5.2 Dynamic Task-to-Core Assignment . . . 35
3.5.3 Runtime Phase Detector . . . 36
3.6 Controller . . . 36
3.7 Practical Factors . . . 38
3.7.1 Limiting Offline Storage Cost . . . 38
3.7.2 Applications without Offline Profiles . . . 40
3.8 Experimental Setup . . . 41
3.8.1 Experimental Platform and Settings . . . 41
3.8.2 Applications with Offline Profiles . . . 42
3.8.3 Applications without Offline Profiles . . . 44

3.8.4 Evaluation Metrics . . . 45
3.9 Results : Single Application per Core . . . 46
3.9.1 Applications with Offline Profiles . . . 46
3.9.2 Application without Offline Profiles . . . 51
3.9.3 Variable Usage Patterns . . . 53
3.9.4 Overhead . . . 54
3.10 Results: Multiple Applications per Core . . . 56
3.10.1 Applications with Offline Profiles . . . 57
3.10.2 Application without Offline Profiles . . . 59
3.10.3 Variable Usage Patterns . . . 62
3.10.4 Overhead . . . 63
3.11 Summary . . . 64

4 Performance and Predictability Issues with Real-Time Trusted Execution 66
4.1 Existing Solutions . . . 67
4.2 Background . . . 68
4.2.1 ARM TrustZone . . . 68
4.2.2 Open Portable Trusted Execution Environment (OP-TEE) . . . 69
4.3 Challenges . . . 71

5 A Real-Time Framework for Trusted Execution on COTS Embedded Processors 72

5.1 Contribution . . . 72
5.2 Related Work . . . 74
5.3 TEE-Task: A Novel Real-Time Task Model . . . 75
5.3.1 Modeling Tasks with TEE Requirements . . . 75
5.3.2 Creating Trusted Execution Segments . . . 78
5.4 Motivations & Problem Statement . . . 79
5.5 T-EDF: A Global Scheduling Algorithm . . . 83
5.5.1 Assigning Jobs to Cores . . . 84
5.5.2 Migrating TEE-tasks . . . 87
5.6 Analysis of T-EDF . . . 88
5.7 Simulations . . . 90
5.8 Experiments . . . 95
5.8.1 Experimental Setup . . . 95
5.8.2 Results . . . 98
5.9 Summary . . . 101

6 Optimized Trusted Execution for Hard Real-Time Applications 102
6.1 Contributions . . . 103
6.1.1 System Model . . . 104
6.1.2 Creating Trusted Execution Segments . . . 105

6.2 Motivation . . . 107
6.3 Offline Super-TEE construction . . . 109
6.3.1 Task Partitioning and TEE Fusions . . . 110
6.3.2 Security Impacts . . . 113
6.3.3 Super-TEE Task Model . . . 115
6.3.4 Feasibility of Super-TEE Candidates . . . 118
6.3.5 Finding Super-TEE Candidates . . . 123
6.4 CT-RM Scheduling Algorithm . . . 124
6.5 Experiments . . . 127
6.5.1 Experimental Setup . . . 127
6.6 Simulations . . . 129
6.7 Related Work . . . 133
6.8 Summary . . . 133

7 A TEE-Aware DAG Scheduling Framework for Hard Real-Time Applications 135
7.1 Contribution . . . 137
7.2 System Model . . . 138
7.2.1 Task Model . . . 140
7.2.2 Modeling DAG-based Parallelism for Secure Execution . . . 142
7.3 T-DAG: Static Scheduling Algorithm . . . 144

7.4 Schedulability Analysis of T-DAG . . . 146
7.5 Evaluation . . . 150
7.5.1 Simulation Setup . . . 150
7.5.2 Simulation Metrics . . . 151
7.5.3 Simulation Results . . . 151
7.6 Summary . . . 152

8 Conclusions 153
8.1 Future Research Directions . . . 154
8.1.1 Extending the Energy Management Framework for Mobile Embedded Systems . . . 154
8.1.2 Extending the Trusted Execution Framework for Energy Constrained Safety-Critical Systems . . . 155

Bibliography 156

List of Figures

3.1 A screenshot of split-screen execution in Android Nougat, where the top window runs Facebook while the bottom window simultaneously runs a web browser. The dimensions of the split-screen can be altered by dragging the black bar separating the two windows. 16

3.2 The proposed framework is decoupled into (1) running applications side-by-side for offline profiling (Step I), (2) analyzing application-specific temporal resource usage pattern (Step II), and (3) an online framework to select the best energy-performance configuration (Step III). 21

3.3 Flowcharts depicting the L1 cache monitor (left) and the LLC cache monitor, with steps for determining the aggregate outbound LLC traffic serviced by the main memory (right). The Cache monitor acquires the PID of target application which runs side-by-side with multiple applications. The profiling script sets up the PMUs to monitor the L1 cache miss events of the target application, and trabsfer the collected data to userspace through Procfs nodes. Similarly, the LLC cache monitor collects the overall system-wide outgoing cache traffic from the LLC to main memory. 26

3.4 Application phase detection using (1) application-specific per-core L1 traffic directed to the shared LLC, and (2) the LLC cache traffic flowing to the main memory. 27

3.5 Memory traffic data directed to and from the main memory from the L2 cache due to YouTube while running side-by-side with the default Android emailing service application on Nexus 6 over time. Each application in the application tuple are dedicated to separate core(s) to isolate the application-specific memory traffic. The data clearly shows periodic surges in data traffic going into the main memory. 29

3.6 Percentage of total execution time spent on different CPU frequency levels (Table 3.1) when using DG and the proposed approach. For CS, a very limited number of CPU frequencies are selected and thus not included to maintain readability. 47

3.7 Percentage of total execution time spent on different memory bandwidth levels (Table 3.1) when using DG and the proposed approach. Note that CS does not adjust memory bandwidth. 48

4.1 Architecture-specific platform security extensions for TEE in CPU, memory subsystem and peripheral storage. 67

4.2 Design framework of the ARM TrustZone software stack showing the flow of execution and data exchange between the secure and non-secure execution environments. 70

5.1 An ordinary real-time task code vs. a TEE-task code which contains sections to be run inside TEE. Step-by-step example shows how to convert a software program into separate executables. 76

5.2 Our real-time task model. 76

5.3 An example illustrating how task dependency and TEE execution are captured by our self-suspending TEE-task model. 78

5.4 Transforming original sensors.c module of PX4 to run in TEE. 80

5.5 TEE section migration example. (Job J_B migrates its TEE section υ_B to core p_1, allowing J_C to begin execution earlier). 82

5.6 Example depicting the calculation of the degree of parallel suspension intervals (DOPS) for job J_B. Both $\text{DOPS}_{B,A}$ and $\text{DOPS}_{B,C}$ are calculated. 86

5.7 Simulation results showing (a), the percentage of feasible task sets as a function of utilization for use case scenario #1 (Table 5.3) (b), the percentage of feasible task sets as a function of utilization for use case scenario #2 (Table 5.3) (c), the percentage of feasible task sets as a function of utilization for use case scenario #3 (Table 5.3) (d), the percentage of feasible task sets of T-EDF as a function of utilization obtained by simulation and feasibility test (use case scenario #3) (e), average task migration count as a function of utilization for T-EDF and G-EDF (use case scenario #3), and (f), the percentage of feasible task sets as a function of core count (use case scenario #3). 92

5.8 (a) A custom-made quadcopter used as a hardware testbed and (b) PX4 flight stack workflow depicting inter-task dependencies according to a message passing protocol. 96

5.9 Code snippet of our representative macrobenchmark that mimics PX4 modules. 100

6.1 (a) Our real-time task model. For a task τ_i without TEE requirement, $\upsilon_i = C_i^2 = 0$; (b) Example offline profiling of the task set (Table 6.6) for super-TEE construction. 105

6.2 Step-by-step example showing how to convert a software program into separate executable. 106

6.3 A motivating example where (a) two tasks need six SMC calls to access two separate instances of TEE execution, but, (b) fused TEE execution sections reduce the number of SMC calls to three. 108

6.4 Code snippet of two example tasks τ_1 and τ_2 which need to be re-factored for TEE execution, and fused into super-TEE and its corresponding trusted segments (TAs). 112

6.5 A example execution instant of the super-TEE (Table 6.3). 118

6.6 A example execution instant of the super-TEE (Table 6.5). 119

6.7 Code snippet of our representative benchmark application modeling the super-TEE shown in Figure 6.4. 130

6.8 Simulation results showing the number of feasible synchronous task sets as a function of system utilization demand with scaling core count. 131

7.1 An example DAG-based real-time task graph. v_s is the source vertex and v_e is the end vertex. The WCETs of each vertex is given within brackets. The complete path $\pi = \{v_s, v_1, v_6, v_9, v_e\}$ is a critical path of the DAG-based task graph. Therefore, the values span(G) and work(G) for the DAG are 16 and 36 respectively. 139

7.2 Our real-time task model. For a task υ_i without TEE requirement, $\upsilon_i = C_i^2 = 0$. 141

7.3 An example DAG-based TEE-aware real-time task graph, formed by transforming a normal DAG-based real-time task graph. The vertices colored green represent the setup time phases of the secure tasks, while the vertices colored yellow represent the rest of the secure tasks. v_s is the source vertex and v_e is the end vertex. A pseudo-source vertex v'_s is used to comply with Property 7.2. The complete path $\pi = \{v'_s, v_s, v_1, v_3, v_e\}$ is a critical path of the original DAG. 143

7.4 Simulation results showing the average application makespan as a function of scaling task nodes with fixed core count (= 4) and secure tasks (= 30%). . . 149

7.5 Simulation results showing the average system speedup as a function of scaling core count with fixed number of task nodes (= 100) and secure tasks (= 30%). 150

List of Tables

3.1 DVFS Configurations for Nexus 6 20

3.2 An Example LUT for Target Application (τ_a) in the Application Tuple (τ_a, τ_b) 23

3.3 Temporal Phase Change Data of Target Application (τ_a) in Application Tuple (τ_a, τ_b) 32

3.4 Nexus 6 Hardware Specifications 41

3.5 Summary of IPC Hits, Energy Savings, and Makespan Increase of Profiled Applications using Proposed Approach Compared to the Default Android Governors (DG) and Critical-Speed Based DVFS Technique (CS) [79] 51

3.6 Summary of IPC Hits, Energy Savings, and Makespan Increase of Applications without Offline Profiles using Proposed Approach Compared to the Default Android Governors (DG) and Critical-Speed Based DVFS Technique (CS) [79] 51

3.7 IPC Hits, Energy Savings, and Makespan Increase of Amazon Music with Varying Application Usage Patterns 54

3.8 Summary of IPC Hits, Energy Savings, Makespan Increase, and Overhead of Spotify using Proposed Approach Compared to Default Governor with Varying Controller Periods 55

3.9 Summary of IPC Hits, Energy Savings, Makespan Increase, and Overhead of Amazon Music using Proposed Approach Compared to Default Governor with Varying Controller Periods 55

3.10 Summary of IPC Hits, Energy Savings, and Makespan Increase of Profiled Applications using Proposed Approach Compared to the Default Android Governors (DG) and Concurrent Workload Classification Technique (WC) [110] 59

3.11 Summary of IPC Hits, Energy Savings, and Makespan Increase of Applications without Offline Profiles using Proposed Approach Compared to the Default Android Governors (DG) and Concurrent Workload Classification Technique (WC) [110] 59

3.12 IPC Hits, Energy Savings, and Makespan Increase of Amazon Music (and Fruit Ninja) with Varying Application Usage Patterns 62

3.13 Summary of IPC Hits, Energy Savings, Makespan Increase, and Overhead of Spotify (with Pokemon Go) using Proposed Approach Compared to Default Governor with Varying Controller Periods 64

3.14 Summary of IPC Hits, Energy Savings, Makespan Increase, and Overhead of Amazon Music (with Fruit Ninja) using Proposed Approach Compared to Default Governor with Varying Controller Periods 64

5.1 Example Task Set 81

5.2 TEE execution overhead 84

5.3 Simulation use case scenarios 93

5.4 PX4 Module List (Higher value means higher priority) 98

5.5 Summary of experimental results of PX4 macrobenchmark (Figure 5.9) running on Rpi 3B using T-EDF and G-EDF with respect to percentage of feasible task sets and average task migration count as a function of task set utilization. 99

6.1 Task Set 1 108

6.2 Task Set 2 117

6.3 Modified Task Set 2 117

6.4 Task Set 3 118

6.5 Modified Task Set 3 118

6.6 Original Task Set 122

6.7 Candidate super-TEE profiles 123

6.8 Optimized Task Set 123

6.9 Summary of experimental results of synthetic benchmark (Figure 6.4) running on Rpi 3B using ct-RM and RM-FF with respect to percentage of feasible task sets as a function of task set utilization. 127

List of Abbreviations

COTS Commercially-Off-The-Shelf

CPU Central Processing Unit

DVFS Dynamic Voltage and Frequency Scaling

IoT Internet of Things

QoS Quality of Service

RTS Real-Time Embedded System

SMC Secure Monitor Call

TEE Trusted Execution Environment

UAV Unmanned Aerial Vehicle

Chapter 1

Introduction

Real-time embedded systems (RTS) are often defined as independent, or a subset of, hardware-software co-design dedicated to performing a specific task, often guaranteed for completion within certain timing constraints. Therefore, performance metrics in RTS not only includes logical and temporal correctness, but also system predictability and design feasibility. Having said that, modern RTS have been armed with more features to support the same (if not more) functionality that have heretofore been limited to general-purpose computing systems. This shift is due, in part, by technological advances, and in part, to satisfy users' increasing demands for more performance, convenience, and versatility. However, RTS are traditionally limited by their SWaP (size, weight and power) constraints. Thus, the challenge lies in optimizing the hardware-software design decisions, to balance the need for improved performance by effectively utilizing the available limited resources, while minimizing system-level energy consumption, with acceptable deterministic timeliness.

1.1 Motivation

This dissertation studies the effect of three major cornerstones of design decisions for resource constrained real-time systems: (1) system-wide power/energy consumption, (2) performance metrics, and (3) timing predictability. Each of these challenges is discussed below, with our proposed solutions.

1.1.1 Power Vs Performance as Design Challenge

The objective of simultaneous execution of multiple applications is to increase productivity and improve quality-of-service (QoS) for end-users. However, such technology results in an increase in power consumption and reduces its battery lifetime of resource constrained embedded systems. Existing voltage and frequency management policies either aim to solely maintain QoS of applications without considering changes in their resource usage pattern, or reduce the power consumption of individual subsystems in an ad-hoc manner. The major challenge in saving energy with an execution context where multiple applications run concurrently is that it is extremely challenging to segregate the dynamic resource usage pattern of individual applications, especially as such applications often have changing resource (e.g., CPU, memory, peripherals etc.) usage requirements over time. The problem becomes even more challenging when the applications utilize the same shared system resources, e.g. memory, at the same time.

1.1.2 Performance Vs Predictability as Design Challenge

Security has become a main concern in embedded systems, which are increasingly interconnected and which often require timeliness. Examples include network node security in

internet of things (IoT), intellectual property (IP) protection in real-time software applications, and dynamic authentication and certification in real-time communication protocols. Hard real-time systems, such as mission critical unmanned aerial vehicles (UAVs), require both privacy/isolation, and strict deadline guarantees. While many existing designs rely on expensive, custom-built hardware with long time-to-market or time-to-deployment to achieve enhanced security, trusted execution environment (TEE) provides an inexpensive and timely alternative for security through platform virtualization by leveraging hardware security extensions. However, TEE cannot be straightforwardly used in real-time systems. Its execution must be judiciously scheduled since its implementation, through secure monitor calls (SMC), attribute to large time overhead and weakened temporal predictability, potentially prohibiting the use of TEE in hard real-time systems.

In this dissertation, we propose three real-time solutions to balance the trade-off between real-time performance and predictable timeliness. In Chapter 5, we present a real-time scheduling framework that leverages the self-suspending task model to capture the unique TEE specifications and inter-task communication requirements, and perform global multicore real-time scheduling within predictable utilization bound. In Chapter 6, we propose super-TEEs, where multiple trusted execution sections are fused together to amortize TEE execution overhead and improve predictability through minimized I/O traffic and reduced switching between normal mode and TEE mode of execution. Finally, in Chapter 7, we present a static task assignment and multicore scheduling framework for real-time trusted execution to amortize the time overhead associated with SMCs, and, improve the temporal predictability of hard real-time systems.

1.2 Major Contributions and Organization

This dissertation extends state-of-the-art design techniques on managing power, performance and predictability in resource constrained real-time systems in the following directions.

1. Due to the vast domain of existing research on managing the power and performance of real-time systems, we limit our observations in this dissertation to smartphones, which is the most popular representative of resource constrained mobile embedded systems. Chapter 2 provides an overview of existing design solutions and the gap in research, along with the major challenges associated with this design paradigm.

2. In Chapter 3, we present an application-aware integrated system-level energy management framework for smartphones that aims to reduce its system-wide energy consumption with negligible impact on QoS of applications whose resource usage characteristics are not precisely known offline or vary over time. The proposed framework has a built-in scheduling policy for assigning side-by-side applications to available cores. Our proposed solution (1) leverages applications' offline profiles to detect instantaneous phase changes (i.e., dynamic changes in resource usage patterns) of the workload of a given application at runtime, and (2) performs adaptive grouping among applications with similar resource usage patterns to pin them onto the same core and execute them simultaneously in split-screen, while (3) dynamically adjusting both the voltage and frequency settings of the processor and memory bandwidth to achieve the most energy-efficient configuration subject to QoS constraints. Our approach is also able to progressively reduce the energy consumption of newly installed real-world applications for which there exists no prior resource usage data, i.e., offline profiles. Experiments on a Nexus 6 smartphone show that our approach achieves an average energy reduction of 15% (13%) and up to 18% (19%) compared to the most closely related work (de-

fault Android governor) for different combinations of real-world applications running side-by-side in split-screen mode. For applications with no prior resource usage data, the proposed framework saves up to 11% (16%) of energy while requiring at most 14 seconds to converge when compared to the most closely related work (default Android governor).

3. Again, due to the vast domain of existing research on managing the performance and predictability of real-time systems, we limit our observations in this dissertation to secure embedded processors. In Chapter 4, we discuss the existing work on extending security in real-time embedded systems, followed by a brief discussion on TEE which is the most popular industry-standard security solution in resource constrained embedded systems. Finally, we list the challenges that exist with this design paradigm.

4. In Chapter 5, we propose our first design solution to balance the trade-off between real-time performance and predictable timeliness. We present a real-time scheduling framework that leverages the self-suspending task model to capture the unique TEE specifications and inter-task communication requirements. In addition, we propose a TEE-aware global scheduling algorithm (T-EDF) for tasks that execute in both trusted and untrusted environments, and compute its worst-case utilization bound. Experimental results on a real hardware platform (Rpi 3B) comparing our algorithm (T-EDF) to global EDF (G-EDF) show an average improvement of 9% in usable utilization, and an average reduction of 52% in task migration counts. A more extensive comparative assessment conducted via simulations also shows an average improvement of 29% and 8% in usable utilization over restricted EDF (r-EDF) and G-EDF, respectively, and an average reduction of 28% and up to 86% in task migration counts compared to G-EDF.

5. Chapter 6 presents our second design solution, super-TEEs, where multiple trusted execution sections are fused together to amortize TEE execution overhead and improve

predictability through minimized I/O traffic and reduced switching between normal mode and TEE mode of execution. Super-TEEs may, however, violate a task's timing requirement and impact the schedulability of the system. We present a technique to enforce the correct timing requirement of a task, along with a sufficient test for schedulability in uniprocessors. We also, discuss ct-RM, a static task assignment and partitioned scheduling algorithm to schedule super-TEEs, alongside other real-time tasks, on multicore systems. Experimental results on a Raspberry Pi 3B, further confirmed by simulations, show that ct-RM outperforms the state-of-the-art technique in terms of usable utilization by 12% on average and up to 27%.

6. In Chapter 7, we propose our third and final design solution. We present a static task assignment and multicore scheduling framework for real-time trusted execution to amortize the time overhead associated with SMCs, and, improve the temporal predictability of hard real-time systems. Our objective is to leverage an existing work, split-TEE task model, to reduce the overall execution overhead of trusted execution in applications where sections of the code that must run in TEE are scheduled to run in parallel with non-secure code sections.

The dissertation concludes in Chapter 8 with some recommendations for future research directions.

Chapter 2

Power and Performance Issues on Resource Constrained Mobile Embedded Systems

We broadly classify energy management techniques for mobile embedded systems into three categories. In the first category, we explore solutions whose aim is to develop an improved power-performance DVFS policy by considering one or multiple components of the system [5, 21, 33, 61, 64, 99, 107, 113]. Li et. al. [77] suggested a general energy management policy that instruments the application code to inform the system of variable workload scenarios, thereby regulating the overall energy consumption of the system. However, it requires that application code be publicly available for modification and/or instrumentation. In the second category, DVFS, dynamic power management (DPM) and/or dynamic thermal management (DTM) are used to develop predictive energy-aware system-level policies [72, 78, 91, 100, 115]. Drawbacks of this type of solutions are (i) backward compatibility issues and (ii) that it requires detailed knowledge of the system P-states, which may be difficult to obtain in proprietary software and devices. Finally, in the third category, the goal is to develop workload characterization techniques to support application-level analysis

and optimization [69, 90, 111, 122, 129]. However, the major limitation of such an approach is the potential lack of support in both hardware and software; existing software profilers either do not capture the fine-grained changes in memory access patterns over time or require code-level instrumentation.

Due to the vast domain of mobile embedded systems, we will be limiting our observations in this work to smartphones, which is the most popular representative of mobile embedded systems. We present key background concepts and review existing work on energy management of smartphones next.

2.1 Background

Android and iPhone devices account for over 90 percent of all smartphones sold worldwide [118]. We chose Android in this paper since (i) it is open-source and, (ii) it employs a flexible Linux kernel at the base of its OS, i.e., Android contains a software stack for mobile devices built on top of a stripped down version of the Linux kernel. In Android, underlying dynamic voltage and frequency management modules are used to manage the energy consumption of the devices. Each CPU-specific DVFS framework contains a set of frequencies the processor can operate at. For instance, Nexus 6's CPUfreq DVFS framework contains 18 CPU frequencies. Similarly, the devfreq framework for memory bandwidth has 13 configurations. A list of available CPU frequency and memory bandwidth, for the Nexus 6, is shown in Table 3.1. The frequency and bandwidth values are interpreted by the OS and device-specific drivers to set the corresponding voltage and frequency on the device. A feasible system configuration is any combination of the available parameters of the tuple (CPU Frequency, Memory Bandwidth).

Google Android [7] is an open-source mobile operating system that is extensively used in

smartphones. Starting with Android Nougat, background applications are not allowed to run at all to aggressively solve the problem of unbounded energy consumption. However, in order to maintain QoS, Android 7.0 and higher supports a split-screen mode where the system fills the screen with two or more applications, (shown in Figure 3.1), showing them either side-by-side or one above the other. The user can drag the dividing line separating the two to make one application larger and the other smaller. This differs from the previous execution model in terms of application context and activity cycle. Previously, Android applications were categorized as foreground and background tasks based on the activity context parameter. While a foreground task enjoys high activity context, the background task is predominantly idle. With the introduction of split-screen however, all the side-by-side executing tasks can demand equally high activity context. A direct result of such a design policy is an increase in energy consumption, overheating, and reduced battery life [9, 10, 11].

2.2 Existing Solutions

Existing Android DVFS policies implemented through device-specific governors [30], such as interactive, ondemand and performance, primarily focus on QoS instead of system-wide energy consumption and, hence, fail to achieve an optimal energy-efficient policy when used in popular applications [26, 109]. In addition, existing solutions are applicable to only a small subset of popular Android applications [34, 71] or performance benchmarks and micro-benchmarks, and were either tested on software simulators or open-source development boards [53, 89, 120].

Rao et.al. [109] establishes the need for an application-specific, performance-aware energy management framework for Android devices, and showed that a coordinated control of system-wide components can save up to 32% energy compared to the default Android gov-

ernors. Similar work by Liang et al. reported that reducing the CPU frequency may not always improve the energy consumption of the system [79]. However, these work either do not consider memory bandwidth management or platforms with multiple applications on split-screen. Reddy et. al. [110] proposes an online energy management solution that performs workload classification for concurrent applications on embedded systems. It uses memory reads per instruction (MRPI) to classify application workload, and then dynamically monitors its performance to select the best CPU Voltage-Frequency setting. However, the proposed solution was not tested on real-world applications, and limited to energy reduction of processor core(s) alone. The work in [93] presented an integrated system-level energy management approach that uses a software support framework to obtain application-specific performance and varying resource usage pattern to find the most energy-efficient CPU frequencies and memory bandwidth without sacrificing QoS. In addition, we detect instantaneous phase changes of applications to further save energy and can be used to minimize the energy consumption of new applications. While the approach aims to reduce the energy consumption of side-by-side applications on smartphones, it fails to dynamically detect phase changes of multiple applications when the latter are running concurrently on the same processor core. Secondly, the work simulates system noise, and spatial and temporal contention of shared resources by running a lightweight application side-by-side with a real-world application.

2.3 Challenges

The challenge is to bridge the gap in existing research by (1) designing an integrated energy management framework that does not require application code to reduce the system-wide energy consumption of side-by-side applications on smartphones while preserving QoS, (2)

enabling further energy savings via the detection of instantaneous phase changes of multiple applications running on each processor core, and (3) conducting experiments on an actual smartphone to assess the energy savings and resultant QoS level of the proposed framework.

Chapter 3

Energy Management of Multiple Side-by-Side Applications in Smartphones

Smartphones continue to have more features and functionality that have traditionally been limited to general-purpose computing systems. This shift was due, in part, by technological advances, and in part, to satisfy users' increasing demands for more performance, convenience, and versatility. To further improve application quality of service (QoS) and/or system utilization, designers have leveraged existing powerful hardware designs, which were originally geared towards general-purpose computing systems, to support multi-threaded, multi-process applications on smartphones. For example, larger dynamic random-access memory (DRAM) in smartphones allow several applications to share the device screen simultaneously. That is, a user could split the screen, viewing a web page on one side while composing an email on the other side of the screen, for instance. However, the main disadvantage of using such full featured hardware design is the increase in power consumption, which reduces battery life and may even cause overheating [10, 11].

Existing energy saving solutions for smartphones often focus on enabling independent fine-

grained power management of distributed components or subsystems at different layers of abstraction [21, 41, 61, 78, 91, 113, 115]. However, independently managing a large number of devices/components is a challenge on a heterogeneous system-on-chip (SoC) design; smartphones often have multi-core CPU, graphics processing unit (GPU), other hardware accelerators such as a digital signal processor (DSP), image signal processor (ISP) etc., and low power DRAM integrated on a single chip. Peripherals may include GPS, modem, wireless local area networks (WLAN), camera, and other off-chip sensors. While optimizing for the power usage of individual devices/components may result in lower power consumption of individual components, the communication power overhead due to data synchronization can be high and must be considered. Moreover, since all components are inter-connected to each other in an SoC, we must be aware of the varying effects of lower power consumption on individual components while determining the overall performance of the system.

Minimizing the power consumption of each SoC component independently does not always result in an optimized solution; since an application utilizes different system components to fulfill each of its diverse functionality, the performance optimization of each SoC module is dependent on the output of many individual modules. For instance, the overall system load is often used to adjust core voltage and frequency settings to optimize application QoS. However, such a technique ignores memory usage, which has been shown to be application-specific [109] and which cannot be easily captured by monitoring system load alone. Since the power consumption of memory subsystems are now comparable to that of processors [119], effective energy-aware designs must not only consider the power consumption due to processor cores but also due to memory and bus subsystems, especially since all the peripheral devices integrated within a smartphone has to rely on the memory subsystem to communicate with the processor, and vice versa.

3.1 Contribution

An effective system-level energy management solution must also consider the varying resource usage pattern of an application over time. Such patterns are difficult to determine offline but can present significant power saving opportunities without sacrificing QoS, as will be shown later. The major challenge is to detect and classify the dynamic resource usage pattern of individual applications within an execution context where multiple concurrent applications run side-by-side on the same shared system resources. We can utilize this information to group applications with similar resource usage patterns, and run them side-by-side on available processor cores accordingly. This can greatly improves the predictability of resource usage patterns, while reducing the runtime overhead of the energy management solution, as reported later in this work. Unlike previous work, e.g., concurrent workload classification by Reddy et. al. [110], which limit the energy saving technique to processor cores only, we propose a coordinated energy management solution that, for the first time, aims to reduce the energy consumption of side-by-side applications on smartphones. The main contributions are as follows:

1. We develop a lightweight phase detection tool, whose front-end resides in userspace while the back-end is implemented in the Linux kernel. The tool records the instantaneous phase changes of concurrent applications, running side-by-side, by leveraging per-core performance monitor unit (PMU) counters, memory access pattern and system bus traffic. This allows us to gain a better understanding of the temporal behavior of an application's system-wide resource usage in order to minimize its energy consumption. Our approach also assists in the energy management of newly installed applications for which no prior resource usage data exists.

2. Given a set of real-world applications, we design an online application-to-core assign-

ment policy which relies on the data collected from our lightweight phase detection tool to search for the best combinations of application tuples with similar resource usage pattern to run concurrently side-by-side on the same core. This helps to judiciously distribute the available shared resources among concurrent side-by-side applications.

3. We construct an application-aware dynamic voltage and frequency scaling (DVFS) framework that jointly reduces the energy consumption of both processor cores and memory subsystems with negligible impact on QoS. The online controller leverages a catalog of offline profiles from the phase detection tool, and selects the most energy-efficient configuration of the system without sacrificing performance of concurrent applications scheduled to run side-by-side on the same core.

4. We validate our proposed approach and assess its performance by comparing it with the most closely related work [110], as well as the default Android governor, in a multi-process environment by running typical real-world applications side-by-side in split-screen mode in Android Nougat on top of a Nexus 6 smartphone.

We implement our proposed solution on Android [7], as it is the most widely used open-source mobile OS. Since the proposed technique does not require application code instrumentation, it is applicable to both open-source and proprietary applications. While we focus on the split-screen technology in this work, our adaptive energy management framework can be applied to other multi-threaded, multi-process application environments. The rest of this paper is organized as follows. Section 3.2 provides an overview of our proposed approach, while the technical details are presented in Sections 3.3-3.6. Section 3.7 discusses practical factors and solutions to address them in our proposed approach. Our experimental setup is discussed in Section 3.8 and the experimental results in Sections 3.9-3.10. Section 3.11 concludes the paper with suggestions for future work.

Figure 3.1: A screenshot of split-screen execution in Android Nougat, where the top window runs Facebook while the bottom window simultaneously runs a web browser. The dimensions of the split-screen can be altered by dragging the black bar separating the two windows.

3.2 Proposed Framework

It has been shown that energy management approaches that independently target components of a smartphone lead to sub-optimal solutions [109]. Similarly, energy saving schemes that ignore changing resource usage of applications fail to optimize the total system energy consumption [79]. Moreover, as previously discussed, multiple side-by-side executing applications can demand equally high usage contexts on available shared system resources (CPU, memory and peripherals), resulting in increased energy profile, overheating, and reduced battery life [9, 10, 11]. An application-specific dynamic phase detection mechanism can monitor the resource utilization pattern of running applications [93, 110]. However, there are multiple challenges associated with its implementation on Androids with split-screen mode of execution. First, we need to segregate, detect, and subsequently predict the dynamic resource usage pattern of individual applications within an execution context where multiple concurrent applications run side-by-side. This helps us isolate the resource utilization of each application from the overall resource contention among concurrent side-by-side applications while running on shared system resources. Secondly, we need to judiciously distribute the limited system resources effectively among concurrent side-by-side applications to achieve system-wide energy savings with negligible impact on QoS. In this work, we overcome these challenges by (1) designing an application-specific phase detection mechanism, which helps in categorizing an application based on their dynamic resource usage patterns, (2) determining the best combination of concurrent side-by-side applications to be run within an integrated energy management framework for minimum system-wide energy consumption with negligible effect on QoS.

An overview of our proposed framework is shown in Figure 3.2. We decouple our work into a three-step procedure; two offline profiling steps followed by online execution. First, we run our target application (say τ_a) through an offline power-performance profiling step

(Section 3.3). We run τ_a side-by-side with another application (say τ_b) to simulate variable system load, and introduce contention on shared resources due to concurrent side-by-side execution. Each choice of τ_b, selected from a list of real-world applications, constructs a different combination of application tuple (τ_a, τ_b). The target application serves as input to the offline profiler to collect relevant individual power usage data and corresponding performance metrics to be used later in online energy management framework. We also catalog offline profile data into categories of applications for the runtime energy management of applications which do not have offline profile data (Sections 3.9.2-3.10.2).

Second, we run the same application tuple (τ_a, τ_b) through an offline phase analysis step (Section 3.4). We capture, over time, the application-specific offline memory traffic using L1 and LLC cache monitors (Sections 3.4.1-3.4.2) and use a phase detector (Section 3.4.3) to generate an offline temporal profile of resource usage pattern (Section 3.4.4) for each target application in the application tuple. We utilize this information to group applications with similar resource usage patterns (Section 3.5.1), and judiciously distribute the available processor cores accordingly using a dynamic task assignment policy (Section 3.5.2). This allows us to improve the predictability of dynamic phase changes during concurrent execution of multiple side-by-side execution online. It also reduces the frequent changes in energy-performance configuration which maximizes their duration of usage of shared resources under similar system configurations, thereby leading to reduced runtime overhead. For our offline profiling steps, we limit the resource contention due to concurrent execution of multiple application by assigning each application in the application tuple exclusively to separate core(s). This design principle helps in isolation of application-specific offline profiles for our target application. For the online step next, we relax this constraint.

At runtime, we use an online controller framework (Section 3.6), which periodically selects the best energy-performance configuration for them, to minimize the overall system-wide en-

ergy consumption with negligible performance degradation. The online controller relies on the resource manager (Section 3.5), along with the offline profiles (Section 3.3-3.4), to select the voltage and frequency levels of associated cores and memory subsystem that minimizes the energy consumption without sacrificing the target QoS. This online control phase is particularly useful for applications that are newly introduced into the system and for which profiles are not readily available. For newly installed applications, we select the most similar profile from the catalog of existing offline profiles by (1) running the newly installed application to analyze its processor utilization characteristics, as well as memory access patterns and, (2) browsing through our list of readily available offline profiles stored in the system to find the closest match to our newly installed application.

In our work, the QoS metric is the makespan of a given application, as it is applicable to both compute-intensive and memory-intensive workloads. Note that other metrics can be used. In addition, since our approach requires a fine-grained QoS metric, i.e., one that can change in a small time interval, to find an energy-efficient frequency and voltage settings for the CPU and memory subsystem, we use the normalized instructions per cycle (IPC), as described later in Eq.3.1, within the framework. While our approach can readily be extended to multiple applications on split-screen, we restrict the number of applications sharing the split-screen to two in this work. This is due to the limitation of our hardware testbed's RAM capacity listed in Table 3.4; recall that all processes sharing the device screen run with equal activity context, resulting in split-screen mode of execution to use extremely high memory in order to keep all the application contexts alive.

Table 3.1: DVFS Configurations for Nexus 6

CPU Freq (GHz)	Level
0.3000	1
0.4224	2
0.6528	3
0.7296	4
0.8832	5
0.9600	6
1.0368	7
1.1904	8
1.2672	9
1.4976	10
1.5744	11
1.7280	12
1.9584	13
2.2656	14
2.4576	15
2.4960	16
2.5728	17
2.6496	18

Mem Bandwidth (MBps)	Level
762	1
1144	2
1525	3
2288	4
3051	5
3952	6
4684	7
5996	8
7019	9
8056	10
10101	11
12145	12
16250	13

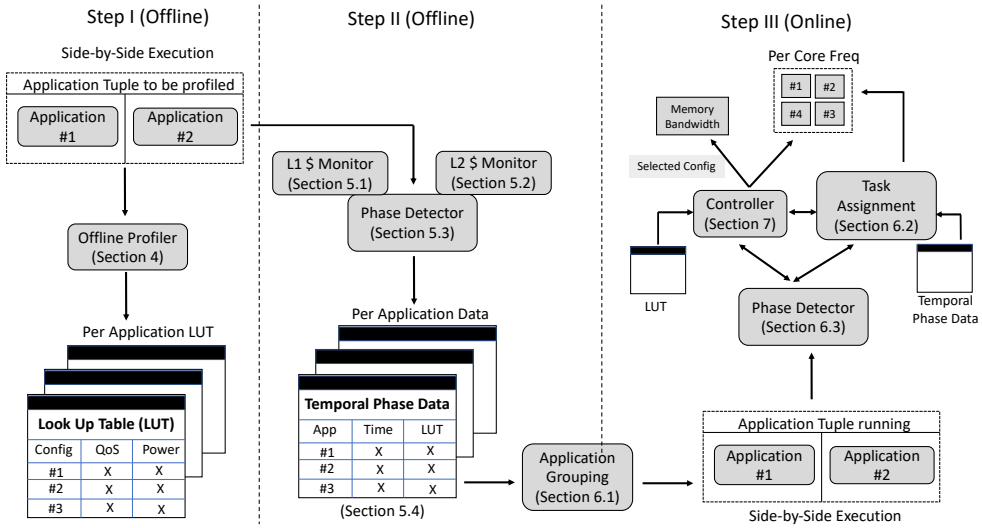


Figure 3.2: The proposed framework is decoupled into (1) running applications side-by-side for offline profiling (Step I), (2) analyzing application-specific temporal resource usage pattern (Step II), and (3) an online framework to select the best energy-performance configuration (Step III).

3.3 Offline Power-Performance Profiling

The first step of our proposed energy management framework is to design an offline power-performance profiler which captures a target application's relationship between its QoS and energy consumption. This information later helps in the online selection of best energy configuration for the system without adversely affecting the performance of the target application. For the collected data to be useful online when multiple applications execute side-by-side, we must account for the competing resource demands by other concurrent applications. Therefore, for each run, we construct a unique combination of application tuples (τ_a, τ_b) from a given set of real-world applications (Sections 3.8.2-3.8.3), where the target application (τ_a) runs side-by-side with another concurrent application (τ_b).

The profiler obtains the average performance data and energy consumption data for a tar-

get application under all CPU frequency and memory bandwidth combinations (Table 3.1). Each such combination, which is represented by the tuple (CPU Frequency, Memory Bandwidth), constitutes a system configuration. A lookup table (LUT), shown in Table 3.2, is constructed for each application and offers information on the relationship among a given system configuration, performance, and power consumption. Each entry of the LUT contains the following parameters: CPU frequency, memory bandwidth, normalized IPC, and energy metric. The energy metric contains the system-wide power consumption obtained while running an application under the corresponding system configuration. As discussed in the previous section, while our overall QoS metric is the makespan of an application, we use the normalized IPC, $\widetilde{\text{IPC}_\text{i}}$, for the i^{th} system configuration (see Table 3.2), inside our framework for fine-grained tuning. The IPC is normalized with respect to IPC_1, i.e., the system configuration with the lowest CPU frequency and memory bandwidth, and defined as,

$$\widetilde{\text{IPC}_\text{i}} = \frac{\text{IPC}_\text{i}}{\text{IPC}_1}, \tag{3.1}$$

where IPC_i is the average IPC of the i^{th} system configuration (Table 3.2).

The problem of large space exploration, however, may impact the practical feasibility of an exhaustive offline data collection process. While the storage cost of an application-specific LUT may be as low as 2 KB, the overall collected offline data set of all combinations of application tuples may exponentially impact its storage cost in resource constrained smartphones. Similarly, the time overhead associated with the offline profile step is another practical caveat to the offline data collection step. We solve them by broadly categorizing each application in the application tuple into computation intensive or memory intensive. In this way, we limit the amount of data that must be stored in the LUT (further explained in Section 3.7.1) while allowing for judicious energy management decisions to be made online. This also assists in the decision making when managing the energy consumption of applications which do not

Table 3.2: An Example LUT for Target Application (τ_a) in the Application Tuple (τ_a, τ_b)

Config	CPU Freq (GHz)	Mem Bandwidth (MBps)	Normalized IPC	Power (mW)
1	0.3000	762	1	2172.9
2	0.3000	1525	1.2	2170.0
3	0.6528	1525	1.3	2288.3
4	0.6528	2288	1.5	1657.6
5	0.7296	2288	1.5	1422.8

have prior offline profile data (Section 3.7.2). The actual number of offline profiles for each set of applications can be selected based on the available memory space on a given system.

3.4 Offline Phase Analysis

The offline power-performance profiler (in Section 3.3) provides a high level trade-off between the QoS of a target application with respect to its energy consumption. However, it fails to record the fine-grained relationship between the application's QoS and its dynamic resource usage pattern. We need this information to determine the best system configuration for an optimal energy management solution. Moreover, the introduction of split-screen mode in Android can cause multiple side-by-side applications executing concurrently to have high interference due to overlapping usage patterns on available shared system resources (CPU, memory, peripherals). Therefore, the challenge is to segregate and detect the resource usage pattern of individual applications within an execution context where multiple concurrent applications run side-by-side while utilizing the same shared system resources at the same time.

We address this challenge by augmenting our offline power-performance profiler, introduced in Section 3.3, with an offline phase analysis step. An offline phase detector (Section 3.4.3)

analyzes the instantaneous phase changes of a target application, along with its CPU load and memory footprint, to (1) understand application-specific behavior and its resource usage pattern, (2) categorize applications based on their resource usage patterns, and (3) distinguish different application phases to target either CPU frequency or memory bandwidth optimization. We broadly classify an application to have a computation intensive phase if, within some time interval, the application spends most of its time on the CPU. Similarly, an application enters a memory intensive phase if, within some time interval, the application spends most of its time fetching/sending data from/into the main memory. The peripheral intensive phase of an application is defined by its temporal use of peripheral devices (e.g., GPU and other hardware accelerators). We limit our phase analysis to only the computation-intensive and memory-intensive phases to determine the resource usage pattern of the target application, as energy management of peripheral devices (1) requires access to proprietary governors and device drivers, and (2) significantly increases both the storage cost of offline profile data and the execution overhead of offline phase profiling. Therefore, we define an instantaneous phase change of a target application as the transition from its computation intensive phase to its memory intensive phase, and vice versa.

An application tuple (τ_a, τ_b), constructed the same way as described in Section 3.3, acts as an input for both the offline phase detector (Section 3.4.3) and the power-performance profiler. However, since concurrent side-by-side applications compete for the same shared resource, we need to isolate our target application's offline profile data from the rest of the system. We do this by pinning our target application (τ_a) to an exclusive core(s), while the other side-by-side application (τ_b) is assigned to the remaining core(s). We leverage the fine-grained measurement method of the load serviced by the L1 private cache and a shared last level cache (LLC) through cache monitors to capture the memory intensive phases of our target application. We also utilize the application-specific per-core CPU activity tool,

Perf [41], over a constant time interval to monitor the percentage usage of the CPU for computation by the application, which is reflected by the IPC value.

3.4.1 L1 Cache Monitor

A major step towards predicting phases of an Android application is to analyze and detect sharp changes in memory access patterns. Our goal is to find out per-core L1 cache misses to account for the average percentage contribution of each core on the aggregate L2 cache outgoing traffic to be serviced by the main memory. The L1 cache traffic is measured in Megabits per second (Mbps). Since Perf does not report on per application per-core L1 cache statistics on Android OS, we leverage the performance monitor units (PMUs) and use them as software event counters. Each PMU counter can be programmed to monitor and register a list of L1 cache events. (Smartphones usually are multi-core systems where each core has a set of PMUs.) In our solution, the per-core per-application data is stored in a hash table which resides inside the global stack of the Linux scheduler. A custom syscall redirects the data to the userspace where we implement our online controller. If such data is made available through proprietary drivers, we could have the framework implemented entirely in userspace. A flowchart depicting the L1 cache monitor is shown in the left side of Figure 3.3. The L1 cache monitor polls the amount of per-core, per-application memory traffic from private L1 caches to the shared L2 cache.

3.4.2 Last Level Cache Monitor

Having recorded the L1 cache traffic, we now need to determine the aggregate outbound LLC traffic serviced by the main memory to predict application phase changes. The LLC miss traffic, along with the application-specific per-core L1 cache traffic, accounts for the average

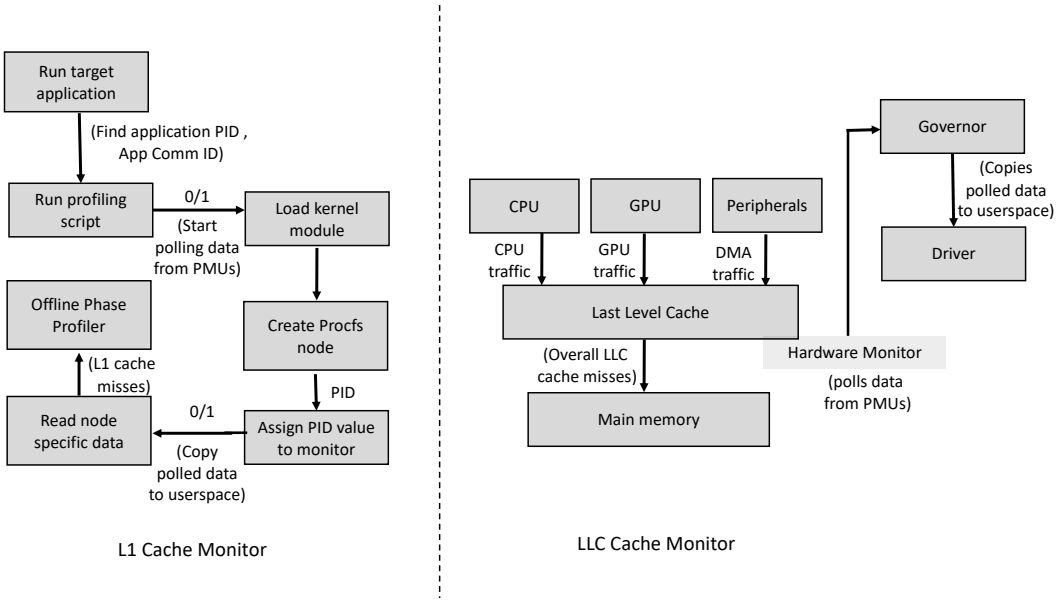


Figure 3.3: Flowcharts depicting the L1 cache monitor (left) and the LLC cache monitor, with steps for determining the aggregate outbound LLC traffic serviced by the main memory (right). The Cache monitor acquires the PID of target application which runs side-by-side with multiple applications. The profiling script sets up the PMUs to monitor the L1 cache miss events of the target application, and trabsfer the collected data to userspace through Procfs nodes. Similarly, the LLC cache monitor collects the overall system-wide outgoing cache traffic from the LLC to main memory.

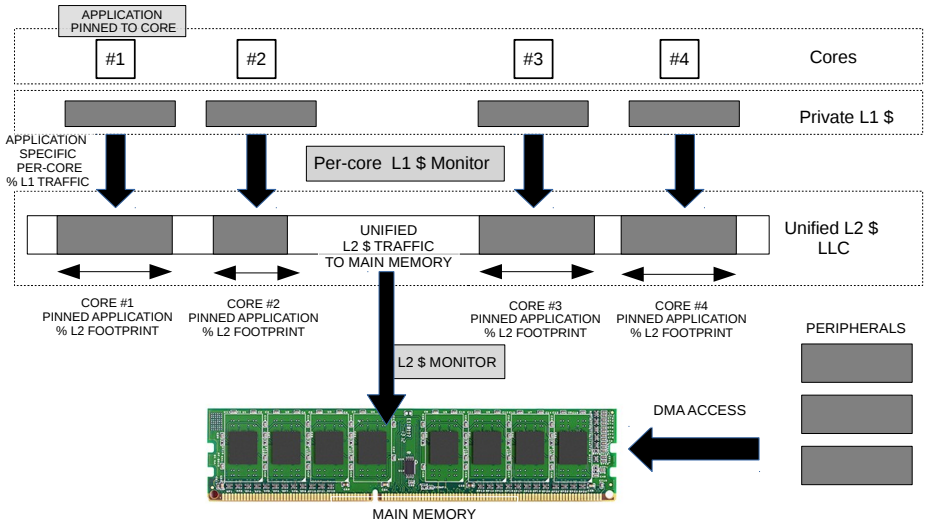


Figure 3.4: Application phase detection using (1) application-specific per-core L1 traffic directed to the shared LLC, and (2) the LLC cache traffic flowing to the main memory.

percentage contribution of each core on the aggregate LLC outgoing traffic. This cache traffic is also measured in Mbps. In our solution, we utilize the system specific hardware PMU (attached to the LLC), and its dedicated driver implementation in the kernel, to export the data related to the LLC activity to userspace for further analyses, as shown in the right side of Figure 3.3. The LLC PMU hardware counter polls every 10 ms for the amount of traffic directed towards the system main memory. Therefore, all data collection mechanisms are also activated every 10 ms. In contrast, the polling frequency of Perf is limited to 100 ms[1], which is inadequate for fine-grained energy management. In addition, while in our case, the Nexus 6's LLC is the L2 cache, our approach can be extended to an arbitrary number of cache levels.

3.4.3 Phase Detector

The objective of the phase detector is to use the dynamic memory traffic data and CPU utilization data to predict instantaneous phase changes of a target application (τ_a) in an application tuple (τ_a, τ_b) running in a split-screen mode. Since running side-by-side applications simultaneously poses a problem of data coalescence on shared resource similar to multi-process execution, we pin each application in the tuple (τ_a, τ_b) to specific core(s) to isolate the memory footprint of individual application. The application-specific per-core data traffic between the CPU and the main memory can now be dynamically monitored by collecting (1) per-core L1 traffic directed to the shared LLC, and (2) the LLC cache traffic flowing to the main memory, as shown in Figure 3.4. We discuss next how to use the data from the L1 and LLC cache monitors, and application specific per-core CPU activity using Perf, to detect the phase change of an application. We also propose a metric to distinguish between a computation intensive phase and a memory intensive phase of an application in question.

Figure 3.5 reports the amount of traffic that is directed to the main memory from the L2 cache due to YouTube while running side-by-side with the default Android emailing service application on Nexus 6. It shows the distinct, periodic memory traffic pattern when we run the application for an extended period of time. Similarly, the CPU activity monitor (using Perf) also shows the CPU intensive phases of the applications. Distinguishing consecutive phases of an application requires the following steps.

We record offline the minimum change in cache traffic that would cause a change in the memory bandwidth under the default Android governor, and quantify it as the sensitivity level (S) of the application under consideration. During offline phase analysis, we use Perf

[1]For the Nexus 6, Perf taps into the PMU framework of the Krait 450 processor through system calls, which is a major reason for its coarse-grained polling intervals.

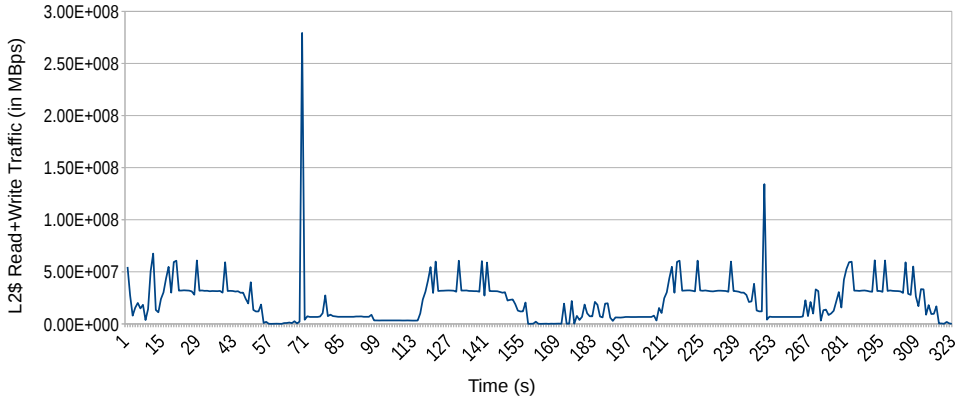


Figure 3.5: Memory traffic data directed to and from the main memory from the L2 cache due to YouTube while running side-by-side with the default Android emailing service application on Nexus 6 over time. Each application in the application tuple are dedicated to separate core(s) to isolate the application-specific memory traffic. The data clearly shows periodic surges in data traffic going into the main memory.

to monitor the CPU intensive phases of the application. Similarly, we use the memory footprint monitors (L1 and LLC cache monitors) over the same time interval to record the application-specific data exchange. We then calculate the variance in application sensitivity level S with respect to both CPU computation and memory traffic as

$$\mathrm{U_j} = \mathrm{S} \times \frac{\mathrm{IPC_i}}{\left(\frac{\mathrm{out\_L1\_traffic}}{\mathrm{out\_L2\_traffic}}\right)}, \tag{3.2}$$

where the resource utilization $\mathrm{U_j}$, $p_j \in \mathrm{P}$, represent the per-core per-application resource utilization of a processor core p_j. If $\frac{\mathrm{U_j}}{\mathrm{S}} \leq 1$, a memory intensive phase of the application is detected. Otherwise, when $\frac{\mathrm{U_j}}{\mathrm{S}} > 1$, a CPU intensive phase of the application is observed. For applications for which no offline profile data exists, we classify them based on the nearest neighbor approach [73] to select the closest matching offline profile. Once we can distinguish the different phases of each application in the application tuple, the next step is to design a way to analyze their temporal behavior.

3.4.4 Temporal Phase Profiling

Given the CPU load and memory footprint of a target application, the phase detector in Section 3.4.3 not only (1) categorizes the instantaneous phases (computation intensive or memory intensive) of the application, but also (2) ascertains the period(s) of duration for each phase. This application-specific temporal phase change data is stored in the form of a LUT where each entry, contains the following parameters: Target App, Side-by-Side App, Temporal Phase Profile, App-Specific LUT. Table 3.3 shows an example phase change data table for a list of real-world applications. While the temporal phase profile stores the time-stamps of beginning and end of each phase of a target application in milliseconds (ms), each entry into the third column stores its power-performance LUT (Table 3.2) introduced in Section 3.3.

In our example temporal phase change data table (Table 3.3), VidCon shows a computation intensive phase between time 0 ms to 15 ms, followed by a memory intensive phase between time 16 ms to 83 ms and so on. The power-performance profile data (collected in the form of a LUT in Section 3.3) is stored in the third column. The M and C associated with each duration window signifies the type of phase the target application is experiencing. Similarly, Facebook has a memory intensive phase between time 0 ms to 66 ms, followed by a computation intensive phase between time 67 ms to 104 ms and so on. Table 3.3 can have multiple entries under a single target application in #App, one for each of its child processes. For example, Facebook application consist of two child processes; (1) user interfacing (UI) and data synchronization of the application, and (2) video streaming functionality. Similarly, we can have an entry for a target application (τ_a) running with different combinations of side-by-side application (τ_b) as selected from a given list of real-world applications. We construct a temporal phase profile for individual applications in the application tuple once when they runs for the first time. The data collected in this step helps in the online resource

management, and minimizing the system-wide energy consumption. Once we have the temporal profile of dynamic phase changes of an application, we can group applications with similar resource usage patterns and run them side-by-side in a split-screen mode as discussed next in Section 3.5.2. For example, the similar phase change patterns of Facebook and MobileBench make them the best candidates for grouping them together into an application tuple for side-by-side execution as shown in our results (Section 3.10).

Note that this step is part of our offline phase analysis technique which runs simultaneously with the offline power-performance profiler. Since we run an exhaustive analyses of system resource usage pattern for all combinations of application tuples from a given set of real-world applications, we will end up with multiple temporal phase profiles of the same target application for different combinations of application tuples. While the storage cost of an application-specific temporal phase change data table may be as low as 5 KB, the overall collected offline data set for all combinations of application tuples may exponentially increase the storage cost in resource constrained smartphones. Again, we address this challenge by broadly categorizing applications as either computation intensive or memory intensive, as explained in Section 3.7.1. Therefore, a computation intensive (memory intensive) application can utilize the temporal phase profile of another previously profiled computation intensive (memory intensive) application. The actual number of phase profile data to be stored offline for each set of real-world applications can be selected based on the available memory space on a given system. This also assists in the decision making when constructing the best combinations of application tuples for which prior offline profile data does not exist (Section 3.7.2).

Table 3.3: Temporal Phase Change Data of Target Application (τ_a) in Application Tuple (τ_a, τ_b)

Target App (τ_a)	Side-by-Side App (τ_b)	Temporal Phase Profile	Target App LUT
MobileBench	Facebook (UI)	{ [M,0,57], [C,58,113] ... }	MobileBench LUT
VidCon	Spotify	{ [C,0,15], [M,16,83] ... }	VidCon LUT
Facebook (UI)	MobileBench	{ [M,0,66], [C,67,104] ... }	Facebook LUT
...	...	...	
...	...	...	
< App Name #1>	< App Name #2>	<Duration#1, Duration#2, ... >	< #N LUT >

3.5 Online Resource Management

Side-by-side execution through split-screen adds to the energy demand as opposed to the traditional foreground-background task execution model since all processes sharing the device screen may run with equal activity context. Heretofore, we have limited the concurrent execution of application tuples (τ_a, τ_b) by assigning each application to specific core(s). This helps in isolating the resource usage (CPU load and memory traffic) of our target application (τ_a) when running simultaneously with another real-world application (τ_b) in split-screen mode. However, in reality, two or more applications may need to share a core when the total number of active applications is large. Similarly, concurrent access to the shared memory can also lead to resource contention. Therefore, we leverage the application-specific offline profile information (power-performance profile and temporal phase change data) to dynamically group side-by-side applications (Section 3.5.2) based on their similar resource usage pattern (Section 3.5.1), and judiciously assign them to the best available core(s) to obtain a minimum energy solution.

Given a set of real-world applications, the objective is to search for the best combinations of application tuples which can run concurrently side-by-side, with minimum system-wide energy consumption while maintaining the application-specific QoS. Let us consider an application tuple consisting of Facebook and MobileBench. Facebook consists of a dominant

memory intensive process, with two other intermittent computation intensive processes. MobileBench, on the other hand, is a representative of an application with multiple phases. We run this application tuple side-by-side in split screen mode. Whenever two or more processes have the similar execution phase(s) (computation intensive or memory intensive), our solution selects the same core(s) for both applications, and separate core(s) otherwise. When two or more processes running on the same core group have computation intensive phase(s), we run them together with a higher core frequency. Conversely, core frequencies are reduced and memory bandwidth is increased during memory intensive phases. Our experimental results show that with respect to the default governor, we achieve an energy improvement of 13.7%. A negligible performance degradation can be attributed to the time multiplexing between two applications at same core(s).

3.5.1 Constructing Compatible Application Tuples

Given a set of real-world applications, our objective here is to group two or more applications with similar phase change patterns into application tuples, and run them side-by-side in a split-screen mode. Previous energy saving approaches [79] either solely concentrated on a single application context or a foreground-background application context where a target application runs simultaneously with a generic lightweight application (e.g., emailing service application) to simulate background interference and resource contention [93, 109]. Our approach, on the other hand, can construct a resource-aware application tuple to execute concurrently in split-screen mode. The motivation behind our approach is to allow applications with similar phase patterns to maximize the duration of usage of shared resource under similar system configurations. This has a two-fold advantage. First, running application processes which are in-phase present a highly predictable almost constant system load pattern, thereby allowing shared resources (processor cores, peripherals) to remain in the same system

configuration (CPU frequency, memory bandwidth) for longer duration(s). Secondly, while reacting to dynamic changes in resource utilization is instantaneous, the process of selected system configuration being reflected at the hardware level incurs a significant overhead (up to 13 ms in our test bed). Therefore, reducing the frequent changes in system configuration leads to reduced runtime overhead, along with improved system-wide power consumption.

We implement a learning-based classification approach, called the nearest neighbor (NN) algorithm [73], for constructing the best application tuples from a list of real-world applications. We leverage the temporal phase change data (Section 3.4.4) available offline for all application profiled a priori to construct the training data set for our proposed classifier. For each candidate application (sayτ_{a}), we use its temporal phase change profile collected offline (in Section 3.4.4) to choose its nearest neighbor application (sayτ_{b}) from the training data set. The application tuple $(\tau_{\mathrm{a}}, \tau_{\mathrm{b}})$, thus constructed, with similar resource usage patterns acts as a benchmark application tuple for runtime energy management. There are multiple advantages of using this classifier in our problem statement. The NN-algorithm has a simple implementation, which is a good criteria for our lightweight energy management tool. It is highly flexible, i.e., new data entry can be easily classified and adapted in real-time into the training data set within a single run. It is also resistant to noisy training data [73]. However, NN-algorithm is inherently slow and do not scale well with increasing size of training data set. We overcome these challenges by (1) bounding the size of the training set by limiting temporal phase profile data (Section 3.7.1) and, (2) storing the data points in two-dimensional tree for faster computation. For an application τ_{a} with offline profiles, we use its temporal phase change data as input to the NN-classifier to find another application τ_{b} with the closest temporal phase change pattern to τ_{a}. The tuple thus constructed forms a resource-aware application tuple $(\tau_{\mathrm{a}}, \tau_{\mathrm{b}})$ for side-by-side execution. Applications which do not have an offline profile can be classified and included in the training data set with

minimum overhead in real-time.

3.5.2 Dynamic Task-to-Core Assignment

Individual applications in an application tuple with similar resource usage patterns have dynamic phase changes which may result in in-phase and out-of-phase execution contexts. An in-phase is defined as an execution context where concurrent applications are in same execution phase (computation intensive, memory intensive or peripheral intensive). Conversely, we denote out-of-phase as an execution context when they are in different execution phase. An in-phase execution context requires same energy efficient system configuration (CPU frequency, memory bandwidth) for all side-by-side applications, while out-of-phase execution context will need different system configurations to maintain QoS of individual applications. Therefore, we implement a dynamic application-to-core assignment policy, which works together with our online controller, to distribute the available resources between concurrently running applications based on their phase contexts, while maintaining the necessary trade-off between minimum system-wide energy consumption and application QoS.

An application tuple (τ_a, τ_b) constructed in Section 3.5.1, and running concurrently side-by-side, is the current workload. Our task assignment tool works in conjunction with the online controller (Section 3.6) to schedule the workload applications to their corresponding core(s) based on their execution context. At each controller period, (1) we compare the current execution phase of each application (τ_a) and (τ_b) from their corresponding entry into the temporal phase change data table collected offline (in Section 3.4.4). We also dynamically detect our workload's instantaneous phase using our runtime phase detector (explained in Section 3.5.3). If both the offline temporal phase and the runtime phase match, we have an in-phase execution context for our current workload. Thus, we assign both applications in

the tuple to the same core(s). If either one of the phases do not match, we split the out-of-phase applications into separate core(s). The task assignment tool also relays the scheduling decision to the online controller. This enables the controller to select the system configuration for each core, and set it according to the current execution phase of the workload for minimum energy solution.

3.5.3 Runtime Phase Detector

Heretofore, we have constructed application tuples with similar dynamic phase change patterns, and introduced a framework to dynamically assign them to the best available core(s) to run them side-by-side in a split-screen mode. This allows us to accurately predict that multiple applications assigned to the same core, running simultaneously, will show similar resource usage pattern. However, we still need to categorize the core-specific instantaneous phase of an application tuple to vote for the best system configuration, and thereby optimize system-wide energy consumption. We leverage the same offline phase detector (Section 3.4.3) to determine the phase change patterns during online execution of the application tuple. As previously discussed in Section 3.4, our runtime phase detector uses the same principle of cache traffic monitoring (Sections 3.4.1-3.4.2) and phase detection (Section 3.4.3) to predict the current phase of the application tuple. The output is fed to the controller, as discussed next, to determine the desired per-core energy-aware system configuration.

3.6 Controller

The goal of the online controller is to attain overall system-wide energy minimization with negligible effect on the application QoS. We construct our online controller based on the

approach taken by Imes et al. [64]. The controller maintains the target performance of an application by leveraging the information from the application-specific system configuration LUTs, the phase change detector, alongside temporal phase pattern. It calculates the current QoS and energy consumption values, and prescribes corrective measures to achieve optimum energy profile.

To maintain QoS, a proportional integral (PI) controller is used. (Based on our results, a more sophisticated controller is not necessary but can be applied.) Our online PI controller provides the output according to

$$\text{co} = \text{co}_{\text{bias}} + k_c \cdot e(t) + \frac{k_c}{t} \cdot \int e(t)dt, \tag{3.3}$$

where co is the controller output, co_{bias} is the controller bias, k_c is the controller gain, $e(t) = |P_{\text{ref}} - P_{\text{measured}}|$ is the controller error, and t is the controller period. The training period (offline) for the controller consists of running each application (for which an offline profile exists) multiple times until the controller selects a configuration that results in P_{ref} on average. This allows us to record the tuning parameter values, k_c and co_{bias}, for each application beforehand.

The controller uses a feedback control loop to achieve the target application-specific performance while working towards an optimized energy configuration of the overall system. The inputs to the controller are (1) the LUT table(s) for a given application tuple (Table 3.2), (2) independent target QoS value(s) to maintain the performance of each application in the tuple executing side-by-side and, (3) the scheduling decisions, along with the relevant execution context information from the task assignment tool. At runtime, the controller is executed as a daemon task while the Android applications of interest run in the foreground in split-screen mode. At the beginning of each controller period t, the controller computes $e(t)$ and uses it

to react accordingly in the next controller cycle. In contrast to the work by Imes et al.[64], we replace the Kalmann filter with our runtime phase detector. The controller output, co, is the required QoS value an application must attain during the current period to maintain P_{ref}. The value co translates to the desired configuration level of the application and is used as a reference to select the most energy-efficient system configuration from the lookup table (Table 3.2). As will be shown in Sections 3.9.4-3.10.4, the overhead of our proposed runtime phase detector and controller is negligible. We now discuss two methods to make our work more practical in the next section.

3.7 Practical Factors

We now discuss two potential limitations to our proposed framework, and present solutions to tackle them.

3.7.1 Limiting Offline Storage Cost

A major concern for a learning-based offline profiling approach proposed by Rao et al.[109], especially on smartphones with limited memory, would be the limitations on the space required to store performance, memory traffic, and energy consumption data for a potentially large number of application combinations which may run side-by-side simultaneously. Dong et al.[45] analyzed the memory access patterns of Android applications and reported the extensive use of native shared libraries among a large number of Android applications. That is, for any pair of applications, one application's shared libraries are often accessed by the other, resulting in similar memory access patterns. These results suggest that by judiciously managing shared libraries, instruction access efficiency as well as the overall performance

can be improved. Our approach relies on these findings [45]. Specifically, we limit the storage requirement of our approach by maintaining a catalog of offline profiles, each of which encompasses a set of applications with similar resource usage patterns (CPU usage, memory, peripheral IP usage, etc.). That is, we classify each set of applications as computation intensive or memory intensive. Computation intensive and memory intensive applications have high CPU utilization and high CPU-memory interactions, respectively. An application is categorized as computation intensive if it has higher number of computation intensive phases than memory intensive phases in its execution context. Similarly, an application is classified as memory intensive if it has more number of memory intensive phases than computation intensive phases in its execution context. The actual number of offline profiles for each set of applications can be selected based on the available memory space on a given system. Note that each entry into the catalog is accompanied by an application's unique identifier (app_comm) and its corresponding reference sensitivity level (S).

We limit the storage cost of offline power-performance LUTs by classifying each application in an application tuple as computation intensive or memory intensive. For a target computation intensive application, we can utilize the NN-algorithm to determine the offline profile of another closest matching computation intensive application which has been profiled a priori. This allows us to save on the storage space that could have been occupied by the target application's LUT. Similarly, for a memory intensive application for which no prior offline profile exists in memory. We leverage the same principle, which is used to limit the amount of LUT (Section 3.3) data, to bound the temporal phase profile data during offline phase analysis of applications. For example, an application which has been categorized as computation intensive (memory intensive) can utilize the temporal phase profile of another previously profiled computation intensive memory intensive application. Since our proposed solution is based on a machine-learning based nearest-neighbour approach, limiting the length of the

training data set also helps in maintaining the computational overhead and scalability of our solution. Moreover, limiting the number of applications in an application tuple to two aides in bounding the maximum achievable size of the training data set. The constraint is due to extremely high memory usage in split-screen mode to keep all the application contexts alive. Therefore, having more than two concurrent applications in split-screen mode is not a viable solution for resource constraint systems with limited battery life.

3.7.2 Applications without Offline Profiles

For applications that are newly introduced into the system and for which profiles are not readily available, we propose to leverage the existing catalog of offline profiles and the online controller to achieve an improved performance-energy configuration. For newly installed applications, we select the most similar profile from the catalog of existing offline profiles by (1) running the newly installed application to analyze its processor utilization characteristics, as well as memory access patterns (and S) and, (2) browsing through our list of readily available offline profiles stored in the system to find the closest match to our newly installed application. We then use the online controller to adapt to the new applications by tuning the controller output to converge to the targeted QoS in the least number of iterations. As will be shown in Sections 3.9.2-3.10.2, an application with no offline profile will only need to run for at most 14 seconds before the proposed framework can achieve significant energy reduction with negligible QoS degradation. Once complete and the controller tuning parameters recorded, later reruns of the application will not require a second tuning, provided the side-by-side execution behavior pattern and system workload do not change.

Table 3.4: Nexus 6 Hardware Specifications

Component	Specification
SoC	Qualcomm Snapdragon 805
CPU	Krait 450 (quad core running at 2.7GHz)
RAM	3GB LPDDR3
Flash	32GB
Sensors	Accelerometer, GPS, Gyro, Baro
GPU	Adreno 420
Wifi	802.11 a, b, g, n, ac, dual-band
Battery	Li-Po 3220 mAh

3.8 Experimental Setup

We next describe our experimental platform, as well as the application benchmarks and evaluation criteria that were used to assess the effectiveness of our approach.

3.8.1 Experimental Platform and Settings

We implemented and tested our energy management framework on a Nexus 6 smartphone, whose hardware specifications are listed in Table 3.4. The Nexus 6 extends a user-level driver support to implement various DVFS policies on the main memory. The Qualcomm Snapdragon chip set is augmented by a piece of hardware attached to the LLC, which serves to monitor real-time LLC traffic, though this feature was not exposed to users. Our implementation exports the hardware-dependent statistics of cache subsystems to userspace. Nexus 6 runs Android Nougat, which introduced side-by-side execution in split-screen mode where multiple applications can simultaneously run with similar foreground context. In our target system, i.e., the Nexus 6 smartphones, the LLC is the L2 cache.

We now provide some specific details for the purpose of reproducibility of results. Before performing offline profiling, the following options were disabled: USB charging to record

accurate system-wide power consumption and mpdecision to prevent CPU-hotplugging. In addition, the CPU_freq_boost configuration flag was disabled to make sure touching the screen does not inadvertently cause the CPU frequency to increase, as this may lead to erroneous data collection. We reduced screen brightness to 50% and the Wifi module was kept on to simulate a common de facto smartphone runtime interference. We construct all combinations of application tuples from the list of real-world applications used in our experimentation (Section 3.8.2- 3.8.3). Our target application tuple run simultaneously side-by-side on the split screen. This allows us to replicate a real-world system environment with application interference on shared resources. In addition, we use Monkey [8] to generate pseudo-random user behaviors in our experiments for reproducibility.

In our setup, the energy consumption is measured using the OS-level battery readings; Android exports the battery status through system-level sysFS nodes, both for instantaneous current (mA) and instantaneous voltage (mV). We deployed a daemon module to record the current and voltage readings to obtain the instantaneous power usage, and subsequently, the system-wide energy consumption during our experiments.

3.8.2 Applications with Offline Profiles

To test our proposed approach on real-world applications, we select a set of five proprietary and open-source android applications as our benchmark suite. We briefly introduce each application next and note its unique features. We construct application tuples consisting of different combinations of these applications to be run side-by-side simultaneously to generate an offline profile, and record it in a LUT as shown in Table 3.2. Next, we run the phase detector to record the temporal phase change patterns of each application, use the data to classify the applications, and construct task tuples consisting of applications with similar

phase change patterns. We then run each task tuple side-by-side in a split-screen mode.

VidCon

A video converter application which uses a specific library to convert videos to different formats. VidCon is primarily a memory-intensive application. It relies less on the CPU and more on hardware accelerators. For our experiments, we converted an mp4 video using the default configurations.

MobileBench

An established browser benchmark. This application shows patterns of stark switches between CPU-intensive and memory-intensive phases. The benchmark loads a collection of website contents onto the phone memory. It offers automatic horizontal and vertical zooming and scrolling as well. The benchmark uses the Chrome browser application to run the tests.

Pokemon Go

A popular Android gaming application. This application shows constant increase in memory bandwidth usage throughout its run with sharp shifts to the GPU-intensive phase. Since this application is GPU-intensive, we have very low CPU usage with very high memory bandwidth usage. The game is manually played for 200 s during our experiments.

Facebook

A popular example of Android social media application which utilizes the system's internet based hardware resources. It consists of three spawned processes, one of which supports the videoplayer feature, while the other takes care of the UI. This application shows sharp increase in CPU and memory usage only when the application is active. We tested the application for 200 s during our experiments.

Spotify

Another popular audio and video streaming Android application consisting of two spawned processes. It shows very high CPU usage with sharp switches to memory-intensive phases during its run. This application is tested for 200 s with songs being changed every 20 s.

3.8.3 Applications without Offline Profiles

Previously, we assumed that we have prior knowledge of the already existing applications (Section 3.8.2) which have their offline profiles stored in the system. Now we consider another set of five applications whose performance-energy profiles are not readily available. Therefore, each new application may or may not have similar resource usage patterns as the applications listed in Section 3.8.2.

1. Media Converter: A video converter application which uses a specific library to convert videos to different formats.

2. Android Browser: The default browser in the Android vanilla OS. The application is used for browsing, viewing images and video playback.

3. Fruit Ninja: Another popular Android gaming application, similar to Pokemon Go mentioned in Section 3.8.2.

4. YouTube: A video sharing application which allows its users to upload, share and view videos online. It offers a diverse category of both amateur and professional video archive.

5. Amazon Music: A popular audio and video streaming application.

3.8.4 Evaluation Metrics

The performance metrics of interest are the resultant QoS level and system-wide energy consumption. As previously discussed, the QoS of an application can depend on a number of factors such as IPC, latency, throughput, etc. Since most Android applications are available as code obfuscated APKs, we focus on IPC and application makespan, as they take into account both CPU and memory performance and are sensitive to system workload. They are also application neutral.

We compared our approach against Android's default governors (DG), as our work is the first to consider the side-by-side execution model. While other governors exist in Android, the default policies were selected since changing governors require root privilege. In addition, the other Android governors are unsuitable for energy-QoS optimization for a number of reasons. Namely, both OnDemand and Performance have the tendency to use the maximum frequency, resulting in excessive energy usage. On the other hand, Powersave tends to select the lowest frequency, thus sacrificing QoS. Interactive, which is the CPU default governor, is more responsive to changing system loads. Similar arguments can be made for device governors. To further validate our approach, we compare against the most closely related work on concurrent workload classification (henceforth referred to as WC) for energy management

on embedded systems by Reddy et. al. [110], and the work by Liang and Lai [79], henceforth referred to as CS, which is a critical-speed based DVFS technique that adjusts CPU frequency but not memory bandwidth.

3.9 Results : Single Application per Core

In this section, we report experimental results pertaining to the benchmark application tuples (τ_a, τ_b), where the target application (τ_a) is running side-by-side with another application (τ_b) set as the legacy Android mailing application. A discussion on the proposed approach is provided next.

3.9.1 Applications with Offline Profiles

For applications that were profiled offline, we report the average value for performance and energy over 10 runs per benchmark in Table 3.5. Figures 3.6 and 3.7 compare the percentage of total execution time spent on different frequency and bandwidth levels between our approach and DG. Note that the frequency and bandwidth levels for CS are not shown to maintain readability, as using CS results in a very limited number of CPU frequencies for a large percentage of time and 100% of time on a single memory bandwidth. In general, our approach, when compared to the default governor, is able to save a significant amount of energy for all applications considered, albeit with small hits in IPC of less than 2% and with an average increase in makespan of up to 3%. In contrast, our approach, shows an overall improvement of 23%, 4% and 6% on energy savings, application IPC, and makespan, respectively, when compared to CS.

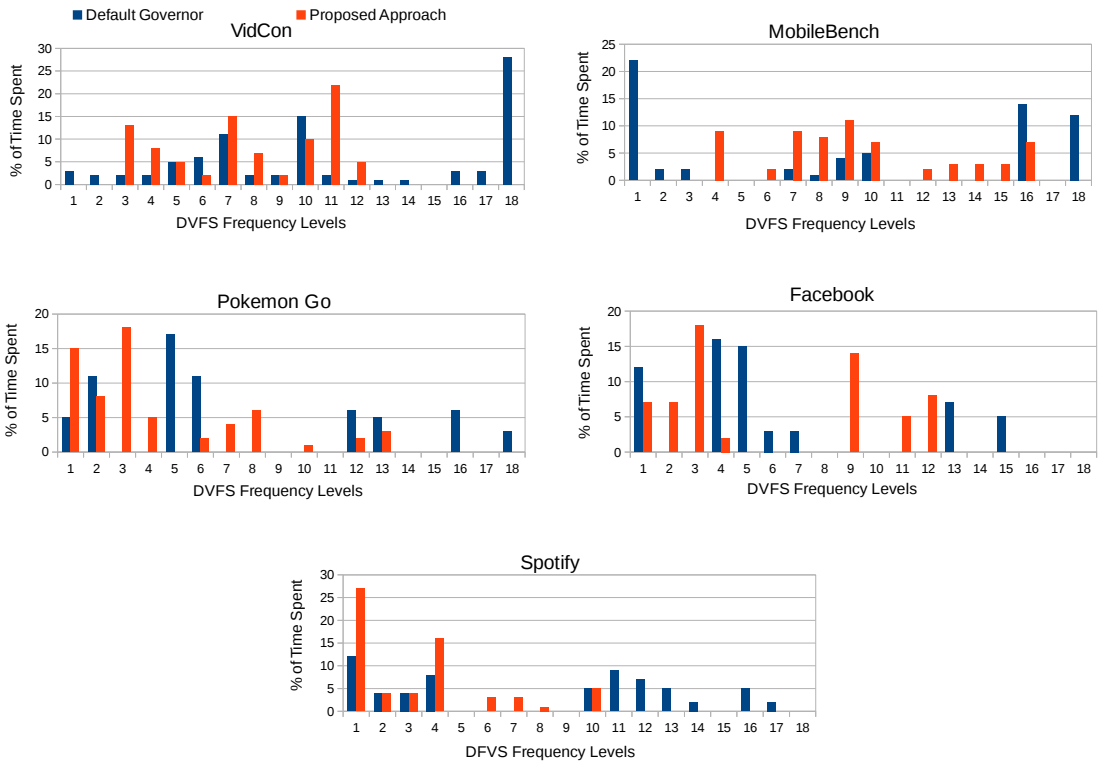


Figure 3.6: Percentage of total execution time spent on different CPU frequency levels (Table 3.1) when using DG and the proposed approach. For CS, a very limited number of CPU frequencies are selected and thus not included to maintain readability.

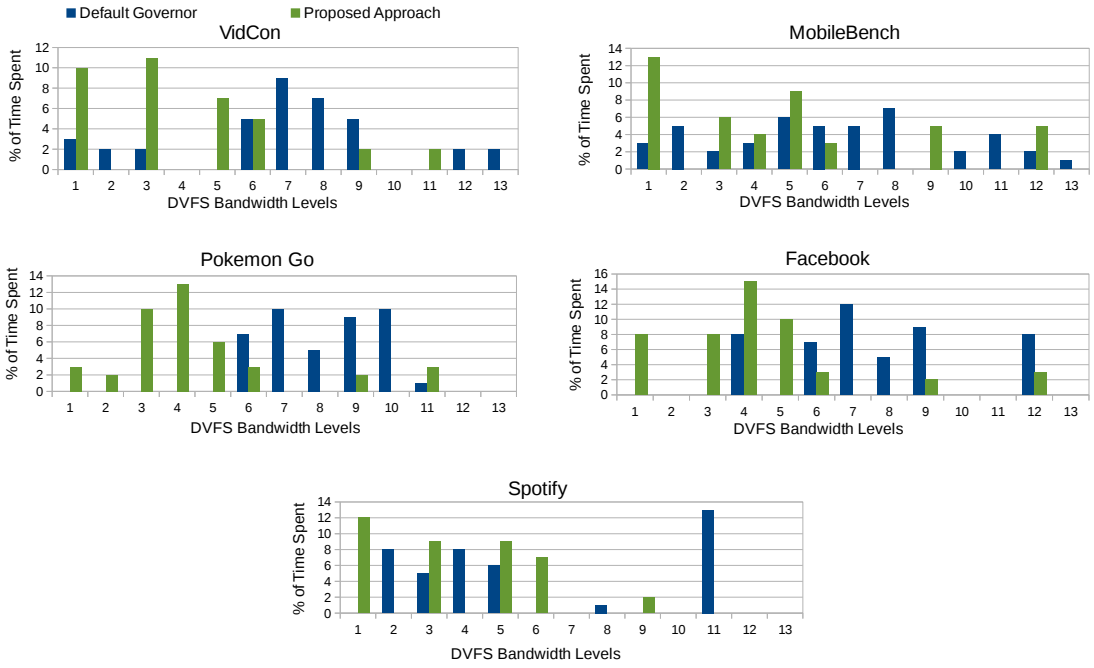


Figure 3.7: Percentage of total execution time spent on different memory bandwidth levels (Table 3.1) when using DG and the proposed approach. Note that CS does not adjust memory bandwidth.

VidCon

As shown in Figure 3.6, using the default governor, the cores spent more than half of the time executing at the highest frequency. This is because DG blindly sets the same frequency for all the cores without considering the individual core loads. This drawback was addressed in our approach, resulting in an energy improvement of 24.5%, IPC hit of 0.2%, and increased makespan of up to 1%. With respect to CS, we see an improvement of 3.6%, 31.1% and 7.2% in IPC, energy savings, and application makespan, respectively. The improvement can be attributed to our holistic approach, which varies both CPU frequency and memory bandwidth.

MobileBench

This application is a representative of application with multiple phases. During a CPU-intensive phase, our solution selects higher core frequencies, as shown in Figure 3.6. Core frequencies are reduced and memory bandwidth is increased during memory-intensive phases (Figure 3.7), resulting in an energy improvement of 18.2% with an IPC degradation of 1.6% and an increased makespan of 6%. When compared to CS, we observe an overall improvement of 0.3%, 11.6% and 1% in IPC, energy savings and application makespan, respectively. The result confirms that the application has more CPU-intensive phases than memory-intensive phases, as this allows CS to optimize the application's critical speed.

Pokemon Go

Since this application is memory- and GPU-intensive (with bursts of high CPU loads), an energy-efficient solution can be obtained by reducing core frequencies while increasing memory bandwidth, as shown in Figures 3.6 and 3.7. Our approach improves the energy

consumption over DG by 19.1% with an IPC degradation of 0.9% and an increased makespan of up to 2%. The slight IPC degradation was due in part by the sharp shifts to CPU-intensive phases, which causes our controller to mis-predict, and, in part, by GPU bottleneck. However, when compared to CS, we see an overall improvement of 5%, 19.6% and 3.2% in IPC, energy savings and application makespan, respectively.

Facebook

From Figure 3.6, our approach selects very different core frequencies compared to DG, resulting in an energy improvement of 7% and IPC degradation of 1.1% with an increased makespan of 2%. Here, less energy can be saved, as Facebook has multiple resource-intensive processes. When compared to CS, we observe an overall improvement of 1.3%, 22.8% and 8.5% in IPC, energy savings and application makespan, respectively. Since Facebook has more memory-intensive phases than CPU-intensive phases, CS fails to optimize the application performance by just calculating the critical speed.

Spotify

Our approach resulted in 27.6% energy improvements with an IPC hit of no more than 1.2% and an increased makespan of up to 3%. Since this application is fairly CPU intensive, core frequencies cannot be further reduced in general (Figure 3.6). However Figure 3.7 shows that the choice of memory bandwidth can be improved to attain both performance and energy gains. In addition, we see an overall improvement when compared to CS of 2.2%, 29.1% and 9.2% in IPC, energy savings and application makespan, respectively. This results highlights a drawback of CS, which adopts an independent, module-based DVFS policy, even though the critical DVFS speed is computed with an informed knowledge of application's memory

Table 3.5: Summary of IPC Hits, Energy Savings, and Makespan Increase of Profiled Applications using Proposed Approach Compared to the Default Android Governors (DG) and Critical-Speed Based DVFS Technique (CS) [79]

Application	IPC hits (%)		Energy savings (%)		Makespan increase (%)	
	DG	CS	DG	CS	DG	CS
VidCon	-0.2	+3.6	+24.5	+31.1	+0.5	-7.2
MobileBench	-1.6	+0.3	+18.2	+11.6	+6.1	-1.0
Pokemon Go	-0.9	+5.0	+19.1	+19.6	+1.4	-3.2
Facebook	-1.1	+1.3	+7.0	+22.8	+ 2.1	-8.5
Spotify	-1.7	+2.2	+27.6	+29.1	+2.5	-9.2

Table 3.6: Summary of IPC Hits, Energy Savings, and Makespan Increase of Applications without Offline Profiles using Proposed Approach Compared to the Default Android Governors (DG) and Critical-Speed Based DVFS Technique (CS) [79]

Application	IPC hits (%)		Energy savings (%)		Makespan increase (%)	
	DG	CS	DG	CS	DG	CS
Media Converter	+0.8	+5.6	+18.7	+22.5	-8.2	-11.1
Android Browser	-0.1	+2.1	+12.4	+13.4	+5.7	-6.0
Fruit Ninja	-3.1	+2.3	-1.4	+15.8	+22.7	-0.7
Youtube	-1.2	+3.0	+15.9	+12.3	+3.1	-2.9
Amazon Music	-2.1	+4.3	+18.3	+20.7	+7.4	-6.8

access rate.

3.9.2 Application without Offline Profiles

To demonstrate the effectiveness of using existing profiles on an application with no offline profile, we performed an additional set of experiments. Here, we assume that the offline profiles for VidCon, MobileBench, Pokemon Go, Facebook and Spotify are readily available. Now, we present a set of newly installed applications, namely, Media Converter, Android Browser, Fruit Ninja, YouTube and Amazon Music, all of which had not previously been profiled offline. Each of the newly installed applications have a unique software signature.

That is, VidCon, MobileBench, Pokemon Go, Facebook, and Spotify are expected to have similar resource usage patterns as Media Converter, Android Browser, Fruit Ninja, YouTube and Amazon Music, respectively. We tested our proposed framework on the new applications using the offline profiles of the existing applications. This idea can be extended to group multiple applications of similar software signatures and matching them to a generic set of unique offline profiles.

Again, we compared our approach against DG and CS, and reported the average value of application performance and energy consumption over 10 runs per application in Table 3.6. From our experiments, compared to the default governor, we found that reusing the profiles of most of the applications helps to save a significant amount of system wide-energy, albeit with small hits in IPC (a little over 3%) and an average increase in makespan of up to 9%. On the contrary, compared to CS, we see an overall improvement in system-wide energy, application IPC, and makespan. We now provide more details on our findings by discussing selected applications (two with the best performance and one with the worst performance) as case studies, when compared with the default governor.

We start with Amazon Music and Media Converter which have the best overall performance when using our proposed solution. Amazon Music has an uneven CPU utilization with one core ending up with very high utilization for a long period of time while the other cores show marginal utilization, with short bursts of high usage. Our solution utilizes the performance-bandwidth offline profile of Spotify for use with Amazon Music. The application reports an IPC degradation of as little as 2% with a 18.3% system-wide energy improvement. On the contrary, compared to CS, the application reports an IPC and makespan improvement of 4.3% and 6.8% respectively, with a 20.7% system-wide energy improvement. Both Spotify and Amazon Music work under a similar client-server module, which likely attributes to their similar CPU usage and memory access pattern. Similarly, Media Converter, which

utilizes the offline profile of VidCon, reports an IPC degradation of as little as 1% with a 18.7% energy improvement when compared to DG. When compared to CS, we see an IPC and system wide energy improvement of 5.6% and 22.5% respectively.

While our proposed approach always outperforms CS, Fruit Ninja shows the worst performance when comparing the default governors with our proposed solution. The primary reason for such QoS degradation can be attributed to a very low degree of shared libraries between the two gaming applications, the other one being Pokemon Go. That was evident from the distinct difference in the memory access pattern of the two applications under consideration. Hence, we observed a negative QoS of 3% and a reduced energy consumption when compared with the default governor's performance. In such a case, we can (i) increase the granularity of the offline profile to improve the phase detection mechanism, or (ii) create an additional LUT at runtime for future use.

To summarize, our approach is able to save a significant amount of energy (up to 18.77%) for almost all the applications considered and which do not have offline profiles, albeit with a hit in IPC of up to 3% and an average increase in makespan of up to 9% when compared to DG. Against CS, we see an overall improvement of 22.5%, 5.6% and 11.1% on energy savings, application IPC, and makespan, respectively. It is interesting to note that CS performs worse than both DG and our approach. However, CS was validated against demo benchmarks and not real-world applications.

3.9.3 Variable Usage Patterns

Most smartphone applications are reactive to user behaviors. In order to validate the robustness of our approach, we experimented on the effect of varying end users' usage pattern on application performance and energy savings. We run Amazon Music (a representative of

Table 3.7: IPC Hits, Energy Savings, and Makespan Increase of Amazon Music with Varying Application Usage Patterns

Application	IPC hits (%)		Energy savings (%)		Makespan increase (%)	
	Monkey	User	Monkey	User	Monkey	User
Amazon Music	-2.1	-2.5	+18.3	+15.6	+7.4	+7.6

applications without offline profiles) under two different scenarios: (i) 5 runs with Monkey [8] to generate pseudo-random interactions with the application; and (ii) 5 runs using an actual user to vary the usage pattern of the application. We report our findings in Table 3.7. The results indicate fairly negligible differences for the different metrics between Monkey and an actual user. Since it is difficult to precisely predict user's behaviors and their specific interactions with an application, we rely on the online controller to adapt to dynamic changes. As future work, we plan on leveraging our framework to incorporate online learning [47] to better predict resource usage patterns over time. Work on hardware-software co-design for application load monitoring [4] can also be applied to our framework.

3.9.4 Overhead

The overhead of our approach depends on the period of the controller (Section 3.6). In this work, said period is set to 1 s, which is a safe lower bound on the controller period, as changing CPU frequency and memory bandwidth takes time on the actual hardware. Combining with online monitoring (less than 10 ms), LUT search (about 7 ms on average), and actually setting new core frequencies and memory bandwidth through sysfs node writes, the total overhead of a single period is no more than 25 ms, which is a little over 4% of the period of the controller. This makes the computation overhead of our approach fairly negligible.

Table 3.8: Summary of IPC Hits, Energy Savings, Makespan Increase, and Overhead of Spotify using Proposed Approach Compared to Default Governor with Varying Controller Periods

Period (in s)	IPC hits (%)	Energy savings (%)	Makespan increase (%)	Overhead
1	-1.7	+27.6	+2.5	4%
2	-1.5	+22.8	+2.4	4%
3	-1.5	+21.5	+2.4	6%
4	-2.9	+9.6	+ 8.0	6%
5	-7.2	+5.0	+11.4	7%

Table 3.9: Summary of IPC Hits, Energy Savings, Makespan Increase, and Overhead of Amazon Music using Proposed Approach Compared to Default Governor with Varying Controller Periods

Period (in s)	IPC hits (%)	Energy savings (%)	Makespan increase (%)	Overhead
1	-2.1	+18.3	+7.4	4%
2	-2.3	+14.7	+7.9	5%
3	-4.1	+11.0	+9.4	5%
4	-6.8	+5.2	+12.2	5%
5	-9.7	+5.0	+17.5	6%

To experimentally assess the overhead of our approach, we select two applications, Spotify (a representative of applications with offline profiles) and Amazon Music (a representative of applications without offline profiles), and compare the performance metrics and energy savings of the applications as functions of the controller period, as shown in Tables 3.8 and 3.9, respectively. We also report the computation overhead herewith. We see a greater degradation in IPC, energy saving and makespan with an increase in controller period over 3 s. This is attributed to our design decision; we collect the CPU related PMU data in hash tables inside the kernel. Similarly, the PMU data recorded for cache traffic resides in a loadable kernel module. Both the data structures are limited by their size and structural integrity. Increasing the controller period beyond 3 s leads to data overflow on the aggregate buffer. A workaround was to truncate the data bits to the first five significant digits, which introduces enough errors and results in fairly poor system configuration decision making. A non-linear growth in computational overhead as a function of the controller period can be attributed to a higher accumulation of online data, which needs to be processed and sorted through. Note, however, that detecting an application's phase change and determining the appropriate energy-efficient configuration take constant time irrespective of the controller period.

3.10 Results: Multiple Applications per Core

In this section, we report experimental results pertaining to the benchmark application tuples (τ_a, τ_b), where the target application (τ_a) and the concurrently running side-by-side application (τ_b) are constructed as various combinations from the set of real-world applications. A discussion on the proposed approach is provided next.

3.10.1 Applications with Offline Profiles

Given a set of real-world applications, that have been profiled offline a priori, our benchmark suite consists of application tuples chosen by the nearest neighbor classification algorithm. Each tuple consists of two real-world applications from our given set where their corresponding temporal phase profiles are most similar to each other. We report on the average value for performance and energy over 10 runs per benchmark tuple in Table 3.10. Our approach, when compared to the default governor, is able to save a significant amount of energy up to 19% for all applications considered, albeit with small hits in IPC of less than 4% and with an average increase in makespan of up to 8%.

VidCon & Spotify

We leverage the dynamic phase detection mechanism (Sections 3.4.3) to collect the data and classify VidCon and Spotify to be a application tuple with closest in-phase characteristics (Sections 3.4.4- 3.5.1). We run this application tuple side-by-side in split screen mode. Since, VidCon is primarily a memory-intensive application, whenever Spotify's memory-intensive phase is in-synchronization with VidCon, our dynamic application assignment framework (Sections 3.5.2) pins both the applications to the same core group, and to different core group(s) otherwise. The default governor, however, performs a naive uninformed application-to-core mapping, which results in most of the cores being spent executing at the highest frequency. This drawback of the default governor was addressed in our approach, resulting in an overall energy improvement of 8.3%, with an IPC hit of 1.8% and 2.4% respectively for VidCon and Spotify. The increased makespan of up to 0.9% and 3.2% respectively for VidCon and Spotify. With respect to WC, we see an improvement of 3.1% (1.5%), 13.9% and 2.4% (9.1%) in IPC, energy savings, and application makespan, respectively for the

combination of VidCon (with Spotify). The improvement can be attributed to our holistic approach, which varies both CPU frequency and memory bandwidth.

Facebook & MobileBench

Facebook consists of three spawned processes, one of which has a dominant memory-intensive pattern, while the others makes use of CPU intermittently. MobileBench, on the other hand, is a representative of an application with multiple phases. We run this application tuple side-by-side in split screen mode. Whenever two or more processes have CPU-intensive phase(s), our solution selects the same core group to run them together at higher core frequency. Similarly, core frequencies are reduced and memory bandwidth is increased during memory-intensive phases. Therefore, with respect to the default governor, this results in an energy improvement of 13.7% with an IPC degradation of 3.6% and 2.4% for Facebook and MobileBench respectively. Both the applications also show an increased makespan of 8.1% and 2.1% respectively. Increased makespan can be attributed to the time multiplexing between two applications at the same core group. When compared to WC, we observe an overall improvement of 1.3% (0.5%), 11.4% and 6.4% (0.8%) in IPC, energy savings and application makespan, respectively for the combination of Facebook (with MobileBench).

Spotify & Pokemon Go

While Spotify is primarily a CPU-intensive application, The popular gaming application Pokemon Go is a GPU-intensive process with hign memory bandwidth. Since both these applications are rarely in the same phase, they predominantly utilize all the core groups to execute side-by-side in split-screen node. Hence, they run as two applications running in separate core groups, with an energy-efficient solution similar to the approach presented in [93].

Table 3.10: Summary of IPC Hits, Energy Savings, and Makespan Increase of Profiled Applications using Proposed Approach Compared to the Default Android Governors (DG) and Concurrent Workload Classification Technique (WC) [110]

Application Tuple	IPC hits (%)		Energy savings (%)		Makespan increase (%)	
	DG	WC	DG	WC	DG	WC
VidCon / Spotify	-1.8 / -2.4	+3.1 / +1.5	+8.3	+13.9	+0.9 / +3.2	-2.4 / -9.1
Facebook / MobileBench	-3.6 / -2.4	+1.3 / +0.5	+18.7	+11.4	+8.1 / +2.1	-6.4 / -0.8
Spotify / Pokemon Go	-2.3 / -1.6	+1.7 / +3.8	+12.5	+18.4	+2.8 / +2.5	-8.1 / -2.7

Table 3.11: Summary of IPC Hits, Energy Savings, and Makespan Increase of Applications without Offline Profiles using Proposed Approach Compared to the Default Android Governors (DG) and Concurrent Workload Classification Technique (WC) [110]

Application Tuple	IPC hits (%)		Energy savings (%)		Makespan increase (%)	
	DG	WC	DG	WC	DG	WC
Media Converter / Amazon Music	-2.5 / -2.4	+4.8 / +3.3	+13.5	+11.7	+1.6 / +4.8	-7.5 / -6.4
YouTube / Android Browser	-2.6 / -0.7	+2.2 / +1.8	+15.2	+8.5	+3.6 / +5.9	-2.6 / -4.1
Amazon Music / Fruit Ninja	-2.3 / -3.6	+4.2 / +1.2	+16.8	+11.2	+4.2 / +15.7	-4.0 / -0.5

Thus, reducing core frequencies of individual core groups while increasing their memory bandwidth improves the energy consumption over DG by 12.5% with an IPC degradation of 2.3% and 1.6% for Spotify and Pokemon Go respectively. We also record an increased makespan of up to 2.8% and 2.5% for Spotify and Pokemon Go respectively. The IPC degradation was due in part by the sharp shifts to CPU-intensive phases, which causes our controller to mispredict, and, in part, by GPU bottleneck. Similarly, with respect to WC, we see an improvement of 1.7% (3.8%), 18.4% and 8.1% (2.7%) in IPC, energy savings, and application makespan, respectively for the combination of Spotify (with Pokemon Go).

3.10.2 Application without Offline Profiles

To demonstrate the effectiveness of using existing profiles on applications with no prior offline profiles, we performed an additional set of experiments. Given a set of real-world applications

(Section 3.8.3) which have not been profiled offline a priori, the objective is to search for the best combinations of application tuples which can run concurrently side-by-side, with minimum system-wide energy consumption while maintaining the application-specific QoS.

Here, we assume that the set of real-world applications, VidCon, MobileBench, Pokemon Go, Facebook and Spotify have their offline profile readily available. Now, we present a set of newly installed applications, namely, Media Converter, Android Browser, Fruit Ninja, YouTube and Amazon Music, all of which had not previously been profiled offline. Each of the newly installed applications have a unique software signature. That is, VidCon, MobileBench, Pokemon Go, Facebook, and Spotify are expected to have similar resource usage patterns as Media Converter, Android Browser, Fruit Ninja, YouTube and Amazon Music, respectively. We tested our proposed framework on the new applications using the offline profiles of the existing applications.

Again, we compared our approach against DG and WC, and reported the average value of application performance and energy consumption over 10 runs per application in Table 3.11. From our experiments, compared to the default governor, we found that reusing the profiles of most of the applications helps to save a significant amount of system wide-energy, albeit with small hits in IPC (a little over 3%) and an average increase in makespan of up to 6%. On the contrary, compared to WC, we see an overall improvement in system-wide energy, application IPC, and makespan. We now provide more details on our findings by discussing selected applications as case studies, when compared with the default governor.

We start with the same approach as with applications with prior offline profiles. We leverage the dynamic phase detection mechanism (Sections 3.4.3) to collect the data and classify Media Converter and Amazon Music to be a application tuple with closest in-phase characteristics (Sections 3.4.4- 3.5.1). We run this application tuple side-by-side in split screen mode. Amazon Music, predominantly a CPU-intensive application, has an uneven CPU uti-

lization with one core ending up with very high utilization for a long period of time while the other cores show marginal utilization, with short bursts of high usage. On the other hand, Media Converter, is a highly memory-intensive application. Therefore, we group application processes with similar phase usage patterns and dynamically pin them to specific core groups to ensure a resource-aware energy profile of the system as a whole. Note, that both Spotify and Amazon Music work under a similar client-server module, which likely attributes to their similar CPU usage and memory access pattern. Similarly, Media Converter, utilizes the offline profile of VidCon. Thus, reducing core frequencies of individual core groups while increasing the memory bandwidth of others improve the energy consumption over DG by 13.5% with an IPC degradation of 2.5% and 2.4% for Media Converter and Amazon Music respectively. We also record an increased makespan of up to 1.6% and 4.8% for Media Converter and Amazon Music respectively. When compared to WC, we see an IPC and system wide energy improvement of 4.8% (3.3%) and 11.7% respectively for Media Converter (with Amazon Music).

When Amazon Music runs side-by-side with Fruit Ninja, it improves the energy consumption over DG by 16.8% with an IPC degradation of 2.3% and 3.6% for Amazon Music and Fruit Ninja respectively. We also record an increased makespan of up to 4.2% and 15.7% for Amazon Music and Fruit Ninja respectively. Fruit Ninja shows the worst performance when comparing the default governors with our proposed solution. The primary reason for such QoS degradation can be attributed to a very low degree of shared libraries between the two gaming applications, the other one being Pokemon Go. Hence, we observed a negative QoS of 3.6% and a increased makespan of 15.7% when compared with the default governor's performance. In such a case, we can (i) increase the granularity of the offline profile to improve the phase detection mechanism, or (ii) create an additional LUT at runtime for future use. However, when compared to WC, we see an IPC and system wide energy

Table 3.12: IPC Hits, Energy Savings, and Makespan Increase of Amazon Music (and Fruit Ninja) with Varying Application Usage Patterns

Application tuple	IPC hits (%)		Energy savings (%)		Makespan increase (%)	
	Monkey	User	Monkey	User	Monkey	User
Amazon Music / Fruit Ninja	-2.3 / -3.6	-2.7 / -3.2	+16.8	+11.5	+4.2 / +15.7	+4.1 / +15.1

improvement of 4.2% (1.2%) and 11.2% respectively for Amazon Music (with Fruit Ninja).

Similarly, Android Browser, when run side-by-side with YouTube, reports an IPC degradation of as little as 0.7% and 2.6% with a 15.2% system-wide energy improvement. Android Browser, utilizing the offline profile of MobileBench, switches between CPU-intensive and memory-intensive phases, providing ample opportunity to run the memory-intensive phases of YouTube in the same core group, thereby optimizing the system configuration. With respect to WC, we see an IPC and system wide energy improvement of 2.2% (1.8%) and 8.5% respectively for YouTube (with Android Browser).

To summarize, our approach is able to save a significant amount of energy (up to 17%) for almost all the applications considered and which do not have offline profiles, albeit with a hit in IPC of up to 4% and an average increase in makespan of up to 6% when compared to DG.

3.10.3 Variable Usage Patterns

In order to test the effect of end users' varying usage behavior pattern on application performance and energy savings, wee experimented on different combinations of application tuples from a given set of real-world applications. Let us consider an example application tuple consisting of Amazon Music and Fruit Ninja running side-by-side in split-screen mode. We run the application tuple under two different scenarios: (1) 5 runs with Monkey [8] to gen-

erate pseudo-random interactions with the application; and (2) 5 runs using an actual user to vary the usage pattern of the application. We report our findings in Table 3.12. Since it is difficult to precisely predict user's behaviors and their specific interactions with an application, we rely on the online controller to adapt to dynamic changes. The results indicate negligible differences in the degree of reaction for the different metrics between Monkey and an actual user, thereby validating the robustness of our approach.

3.10.4 Overhead

The proposed approach in this paper can be divided into an offline phase and an online phase. While the overhead associated with the offline process is not a bottleneck, the online process, however, is bounded by the period of the controller (Section 3.6). In this work, said period is set to 1 s, which is a safe lower bound on the controller period, as changing CPU frequency and memory bandwidth takes time on the actual hardware. The overhead associated with our online process can be attributed to (1) online phase monitoring (less than 10 ms), (2) instantaneous phase classification and application-to-core assignment (less than 3 ms), (3) LUT search (about 9 ms on average), and (4) actually setting new core frequencies and memory bandwidth through sysfs node writes. Therefore, the total overhead of a single period is no more than 35 ms, which is a little over 4% of the period of the controller. This makes the computation overhead of our approach fairly negligible.

The solution proposed in WC by Reddy et.al. [110] implements a reactive controller to select the optimal energy efficient CPU voltage-frequency level. However, their reactive index, i.e., the instantaneous changes in system-wide resource utilization is preset at 1%, which means that an increase in memory reads per instruction (MRPI) data over 1% will trigger a change in the CPU voltage-frequency level. Since it is hard to predict the overhead associated

Table 3.13: Summary of IPC Hits, Energy Savings, Makespan Increase, and Overhead of Spotify (with Pokemon Go) using Proposed Approach Compared to Default Governor with Varying Controller Periods

Period (in s)	IPC hits (%)	Energy savings (%)	Makespan increase (%)	Overhead
1	-2.3 (-1.6)	+12.5	+2.8 (+2.5)	4%
2	-2.3 (-1.9)	+12.5	+2.9 (+3.3)	4%
3	-2.1 (-2.6)	+11.2	+2.8 (+3.2)	6%
4	-4.9 (-6.1)	+9.2	+4.4 (+7.8)	6%
5	-6.0 (-6.9)	+9.1	+7.6 (+9.1)	7%

Table 3.14: Summary of IPC Hits, Energy Savings, Makespan Increase, and Overhead of Amazon Music (with Fruit Ninja) using Proposed Approach Compared to Default Governor with Varying Controller Periods

Period (in s)	IPC hits (%)	Energy savings (%)	Makespan increase (%)	Overhead
1	-2.3 (-3.6)	+16.8	+4.2 (+15.7)	4%
2	-2.4 (-3.7)	+16.8	+4.2 (+15.9)	5%
3	-4.5 (-4.9)	+12.6	+9.5 (+16.2)	5%
4	-7.2 (-6.3)	+11.0	+18.1 (+15.3)	5%
5	-11.4 (-7.4)	+7.3	+18.5 (+18.0)	6%

with each controller period for WC, we do not compare it with our proposed approach across varying controller periods. However, for a controller period of upto 2 s, we report an overhead of 3% for the controller implementation used in the WC approach, along with greater degradation in IPC, energy saving and makespan.

3.11 Summary

In this article, we presented an holistic system-level energy management solution for resource constrained embedded systems, especially smartphones. We designed a fine-grained lightweight offline profile-based tool to capture application-specific performance and mem-

ory traffic data, and detect instantaneous phase changes of applications. We leverage the collected data to monitor multiple applications online which run concurrently side-by-side in split-screen mode, while utilizing the shared system resources. An online controller leverages application-specific temporal phase change profiles to find the most energy-efficient CPU frequencies and memory bandwidth without sacrificing QoS. Experiments on a Nexus 6 smartphone on real-world Android applications validate the feasibility of the proposed approach. For future work, we propose two broad research problems. While we limited our energy management to CPUs and memory subsystem, we propose to include peripheral devices within the purview of our integrated framework. This requires tapping into device-specific governors and proprietary drivers. Another direction of research includes analyzing the relationship between other classification algorithms and the granularity of task grouping, and thereby its effect on the underlying energy consumption.

Chapter 4

Performance and Predictability Issues with Real-Time Trusted Execution

As real-time embedded systems become more complex and interconnected, certain sections or parts of the systems may need to be secured to prevent unauthorized access, or isolated to ensure correctness. In the case of hard real-time systems there is a need to provide security while maintaining strict real-time deadlines. Let us consider a surveillance unmanned aerial vehicle (UAV) as an example. The UAV must perform sensitive operations, such as obtaining coordinates of interest for surveillance. However, an attacker may learn how the UAV communicates with ground control, and take control of the UAV, along with the sensitive information it carries Once the attacker has access to the communication channel between the on-board computer of UAV and the ground station, it can direct the UAV to either broadcast or corrupt the sensitive information. Therefore, it is crucial to protect such sensitive information from access by an unauthorized party. In such cases, we must isolate sensitive information from the rest of the system to prevent such inadvertent leaks to an unauthorized party.

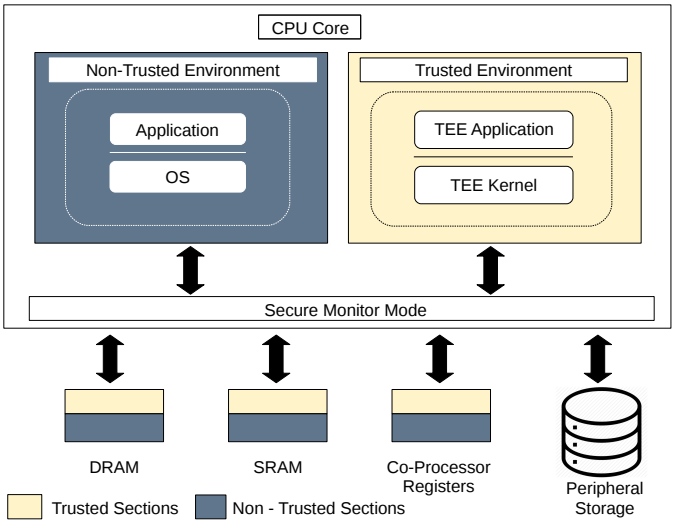


Figure 4.1: Architecture-specific platform security extensions for TEE in CPU, memory subsystem and peripheral storage.

4.1 Existing Solutions

Existing approaches to protect hard real-time applications include hardware-based encryption, isolated execution using additional hardware, and code obfuscation [28, 37, 128]. However, they all require expensive, custom-built hardware with long time-to-market or time-to-deployment cycle. A readily available alternative is the use of trusted execution environment (TEE) on commercial off-the-shelf (COTS) embedded processors. Examples of hardware support for TEE include the ARM TrustZone [12] and Intel Trusted Execution Technology (TXT) [55]. TEE leverages hardware security extensions to provide platform virtualization to run an application in secure isolation from the rest of the system. TEE can be quickly redeployed if an exploit is found, as they are implemented using platform virtualization and, hence, would not require hardware redesign. Virtualization itself is known to have many security benefits including isolation and restricted resource sharing [126]. In TEE, certain sections of the hardware, such as memory, registers and peripherals, are made available only

to the TEE. TEE is utilized by partitioning code such that the portion that is related to secure execution is run inside the TEE and communicates with the rest of the code via a hardware controlled channel. Hence the applicability of TEE encompasses industry-standard certification, security and isolation. For example, in the case of IP protection, TEE can be used for sand boxed execution of firmware or software that include algorithms, data etc., which, if compromised, negatively impacts a system's confidentiality. Similarly, for real-time security management in IoTs, sensitive information can be hidden from the rest of the network by storing it in encrypted memory space and accessed through layers of authentication and matching certification.

4.2 Background

Relevant background materials on the trusted execution environment (TEE), with a focus on the most popular industry-standard technology are described in this section.

4.2.1 ARM TrustZone

The ARM TrustZone [12] is an embedded and secure virtualization platform for COTS embedded systems. Platform virtualization enables enhanced system security through either (i) a TEE or, (ii) a trusted platform module (TPM). TEE consists of a virtualized environment where code that requires trusted execution can be securely executed in isolation from the entire system. Figure 4.1 shows two separate execution environments, namely normal world (running the non-trusted OS) and secure world (running the trusted OS). The trusted OS, in comparison to non-trusted OS, has limited hardware/software features, and reduced functionality. The TEE and non-TEE cannot simultaneously execute on the same processor

core since the processor can be in one mode only at any given time, but can run in parallel on separate cores. Communication between TEE and non-TEE environments during a mode switch is established through an architecture-specific secure message passing protocol.

In a non-trusted execution environment, the non-secure applications, i.e., those that do not require TEE execution, run on standard embedded hardware resources. In a trusted environment, all trusted execution code and its data are stored in, copied to, and executed in isolation on specially augmented secure hardware resources, e.g., CPU, memory, and peripherals. These secure hardware extensions provide data encryption to protect the secure environment data/code from being accessed by the non-trusted software. The trusted OS may also logically deactivate a subset of existing secure peripherals in the respective execution environments in order to minimize the probability of unauthorized access on unprotected peripherals. TrustZone software delimits the code running in the normal world from changing the secure OS system state. This means that if an attack is routed through the normal OS, it is confined to the access privileges of only the non-secure environment. On the other hand, the trusted OS is vulnerable to internal attacks. A security breach can happen through the secure OS if a trusted application chosen to run on TEE is inherently malicious. Hence, it is the programmer's responsibility to ensure that the trusted applications do not inadvertently introduce any security vulnerabilities.

4.2.2 Open Portable Trusted Execution Environment (OP-TEE)

OP-TEE [2] is an open-source TEE implementation by Linaro to integrate ARM-compatible standard Linux with ARM TrustZone. It uses the standardized GlobalPlatform TEE specifications [1] to construct a framework for ARM TrustZone to co-exist with a standard Linux distribution. Figure 4.2 shows the detailed setup of an OP-TEE port into a Linux environ-

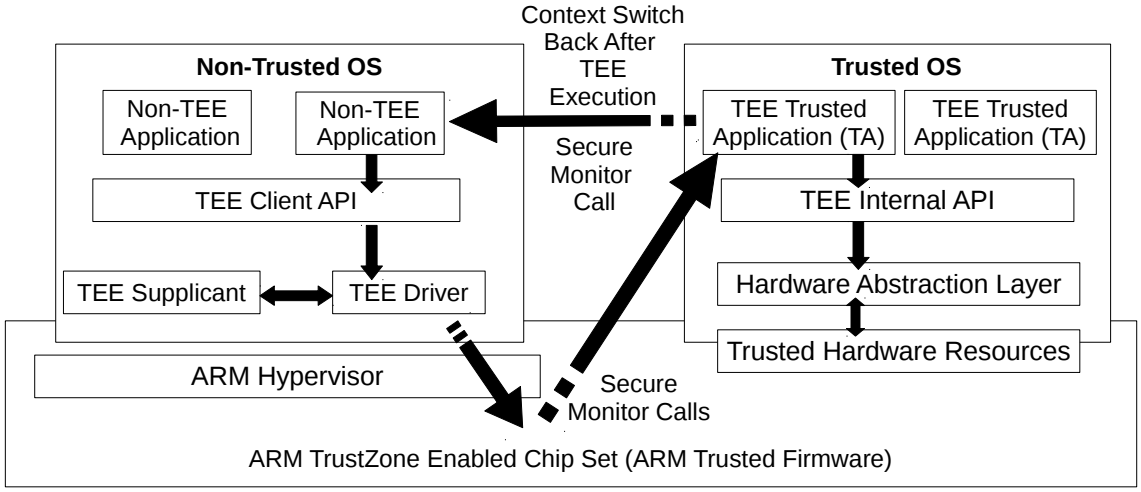


Figure 4.2: Design framework of the ARM TrustZone software stack showing the flow of execution and data exchange between the secure and non-secure execution environments.

ment. The OP-TEE OS consists of three main components, (i) a hardware abstraction layer which forms the backbone of platform virtualization and provides a communication channel with the non-trusted OS, (ii) a minimal secure kernel and its associated TEE internal APIs to support trusted execution and (iii) a set of trusted applications which can run in isolation on OP-TEE OS. Similarly, the co-existent non-trusted OS is augmented with a secure interface to communicate with a trusted application.

Non-secure applications use the TEE-client APIs exposed by the OP-TEE driver implemented in the Linux kernel. When a non-secure application requests for TEE, the driver intercepts the SMC, suspends the calling application and triggers an SMC handler in the OP-TEE OS. The OS then loads the required TA in a secure kernel thread, executes the TA and switches back control to the non-trusted OS. When an interrupt comes from either secure or non-secure world, OP-TEE OS saves the TA context and suspends it, triggers the interrupt handler (or switches to normal world and triggers the handler there if it is a non-

secure interrupt), reloads the TA context and continues execution. Note that a TA need not continue its execution on the same core after reload.

The OP-TEE OS implementation does not have a process scheduler. Hence, it uses the process scheduler of the host OS to run its secure kernel thread. However, OP-TEE OS maintains an active secure thread stack to service non-secure interrupts, thread migration and premature process termination. Note that the trusted OS reserves the right to define and service its security specific implementations. For more details regarding OP-TEE, readers are referred to the GlobalPlatform specifications [1] and OP-TEE project website [2].

4.3 Challenges

Utilizing TEE creates multiple challenges from a real-time perspective. First, trusted execution introduces additional time overhead, which could cause deadline misses if such an overhead is not taken into consideration [85]. Specifically, each instance of TEE execution (in ARM TrustZone) is initiated by a setup phase and exits through a destroy phase. Since TEE leverages architecture-specific secure monitor calls (SMC) to realize these phases, the time overhead associated with TEE execution, including setting up and tearing down a TEE session, requires 18,500 μs on a Raspberry Pi 3B (Rpi 3B) (Table 5.2). Second, trusted execution may adversely affect the deterministic execution of the system, as tasks running inside a TEE may need to communicate with other tasks that are executing on the native real-time operating system. Similarly, data/instruction fetches and writebacks during TEE execution is a major cause for large variation in task execution times, weakening predictability.

Chapter 5

A Real-Time Framework for Trusted Execution on COTS Embedded Processors

Utilizing TEE creates multiple challenges from a real-time perspective. First, trusted execution introduces additional time overhead (experimentally reported in Table 5.2), which could cause deadline misses if such an overhead is not taken into consideration [85]. Second, trusted execution may adversely affect the deterministic execution of the system, as tasks running inside a TEE may need to communicate with other tasks that are executing on the native real-time operating system. Thus, there is a need for a new task model that can capture the complex relationship among the tasks inside and outside of a TEE to permit schedulability analysis.

5.1 Contribution

we adopt the self-suspension task model [32] to model real-time tasks requiring trusted execution. We also present a novel task assignment and scheduling framework for real-time

trusted execution on COTS processors. In our framework, code or applications that contain sensitive information are executed in isolation from the rest of the applications without sacrificing timeliness. Since this work solely focuses on the modeling and scheduling of hard real-time tasks which require trusted execution, and does not require modifications to the operations of TEE, the security guarantees and benefits provided by TEE remain unchanged. Readers are encouraged to refer to existing work [56, 62, 130], which examines how TEE can improve the security of a system. Our main contributions are as follows.

1. We present a general real-time task model, which is based on the self-suspending task model, to capture the trusted execution requirements of dependent hard real-time tasks.

2. We develop a global real-time scheduling algorithm called T-EDF to schedule the tasks with trusted execution requirements on a multicore platform. In addition, we derive a sufficient condition for schedulability of the proposed T-EDF algorithm.

3. We implement and validate our approach on a quadcopter with a Raspberry Pi 3 using ARM TrustZone [12] and OP-TEE [2]. Since existing autopilot software does not support TEE (see section 5.8 for details), we create a macrobenchmark which captures the timing and dataflow requirements of the open-source PX4 flight stack [3] with secure sections running inside a TEE.

4. We extensively assess the pros and cons of our proposed approach in comparison to the state-of-the-art technique in a custom-built simulated environment.

5.2 Related Work

Existing work that lies at the intersection of security and real-time systems often focus on the trade-off between real-time constraints and security levels [60, 67, 86, 98]. For instance, Ahmed and Vrbsky [6] showed the generation of covert channels when real-time database access is not handled in a timely manner for tasks having different levels of security and provided concurrency control methodologies to maintain real-time behavior and security. Jiang et al. [68] used FPGA co-processors to balance the energy consumption against real-time deadlines and security requirements. Other work have used task scheduling to maximize the security level of a given system subject to real-time constraints [80]. Hasan et al. [59] created a security enforcing server that executes sporadically in a fixed-priority system. In this work, we explore the feasibility of using TEEs for trusted execution in hard real-time systems. As such, our work is complementary to these existing work.

The ARM TrustZone has been the platform of choice for many security and embedded systems. For instance, hypervisors implementations [50, 106] allow non-trusted execution environments such as Linux OS, and trusted execution environment, to be run simultaneously while providing convenient performance measurement tools. While the work in [105] presents a case for TrustZone in real-time internet of things (IoT), Martins et al. [87] developed a parallel implementation of TrustZone to increase its real-time support and improve its degree of compatibility to run alongside unmodified guest OS. Commercial implementations, such as Samsung Knox [70] provide a container based solution for executing trusted code separately from the Android operating system. KNOX utilizes the ARM TrustZone to store confidential data. However, KNOX is built into Samsung's mobile embedded devices and its usage is limited to these devices. Pinto et al. [104] used the ARM TrustZone by means of a TEE running as a low priority thread of a real-time OS. None of the existing work considers real-time deadline constraints with TEE execution as the primary requirement.

While existing work use a proprietary, non-standardized method to access ARM TrustZone, we use a standardized, open-source portable approach.

OP-TEE [2] is another open source framework for ARM TrustZone. It differs from the others in that it is based on the GlobalPlatform specifications [1] and its goal is to provide a standardized way to access and use the TEE. Liu et al. [84] proposed a system which runs trusted sections in an OP-TEE controlled TrustZone environment alongside a real-time Linux environment. However, it ignores the overhead associated with OP-TEE and its impacts on real-time deadlines.

5.3 TEE-Task: A Novel Real-Time Task Model

In this section, our goal is to present a hard real-time task model that can (1) be used to capture the trusted execution related requirements of a real-time task, (2) express data communications among the tasks and (3) be used for assignment and scheduling. We will leverage the self-suspending task model to accomplish this goal. We also provide an example that shows how a software program can be transformed into a task with TEE requirements.

5.3.1 Modeling Tasks with TEE Requirements

We begin by distinguishing between a real-time task with no trusted execution requirements (i.e., ordinary real-time task) and a real-time task that requires trusted execution. Figure 5.1 shows a representation of an ordinary real-time task on the left and a task with trusted execution requirements on the right. Henceforth, we will refer to a task with trusted execution requirements as a TEE-task. Formally, an ordinary periodic real-time task τ_i is described by the following tuple: (C_i, T_i), where C_i is its worst case execution time and T_i is its minimum

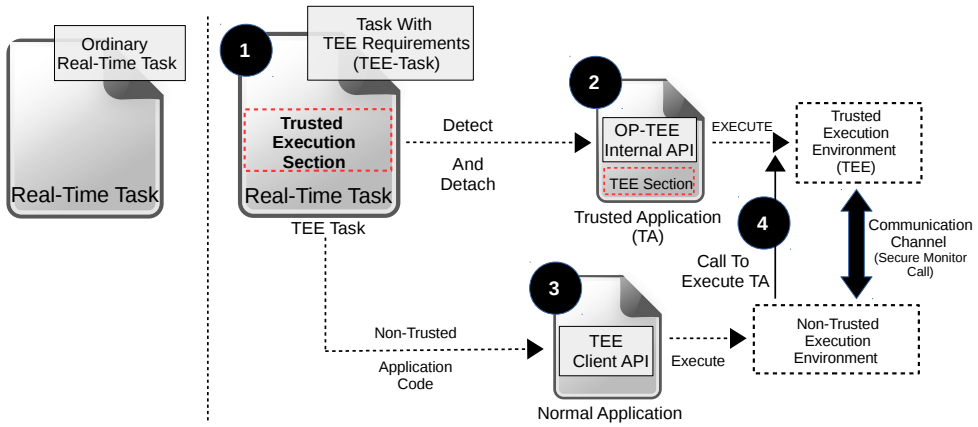


Figure 5.1: An ordinary real-time task code vs. a TEE-task code which contains sections to be run inside TEE. Step-by-step example shows how to convert a software program into separate executables.

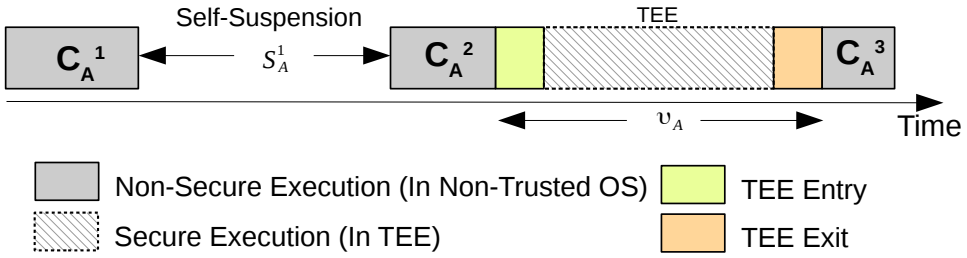


Figure 5.2: Our real-time task model.

inter-arrival time. Unless specified otherwise, deadlines are equal to periods. In addition, when the context is clear, an instance of a task τ_i is referred to as J_i.

Formally, we describe TEE-task, τ_i^{t}, $i = 1, \ldots, n$, by the following tuple: $(T_i, C_i^1, \upsilon_i, C_i^2)$, where T_i is as previously defined, and two non secure computation segments (C_i^1, C_i^2) are interleaved by a single trusted execution interval (υ_i) of τ_i^{t} [1]. The upper bound on non-secure computation segment is $C_i^1 + C_i^2$. The upper bound on trusted execution part of τ_i^{t}, υ_i, includes TEE entry, secure memory copy, and TEE exit as shown in Figure 5.2.

[1]Though we consider a single trusted execution interval for simplicity, it is not advisable to have multiple TEE execution sections interspersed with non-secure computation segments within a TEE-task, since each trusted execution incurs exceptionally high timing overhead (Table 5.2).

Since data communications among tasks are not allowed inside TEE for security reasons, a TEE task τ_i^{t} must prefetch its data/code before entering TEE. Hence, data dependencies among the tasks must be modeled. For instance, our example real-world application (PX4) [3], described in Section 5.8, performs data synchronization using a producer-consumer inter-task data dependency model. Both lock-based and lock free data synchronization protocols incur additional computational overhead due to "retry loops" to gain access to a shared resource [43]. Let B_i represent the upper bound on the retry loops used to gain access to a shared resource for a dependent task τ_i. We assume that the value for B_i is supplied as part of the task dependency graph (Figure 5.3). We also leverage the self-suspension task model [32] where each self-suspension interval represents the maximum time during which a task is blocked for data, and which always precedes the trusted execution interval. A dependency graph such as the one shown in Figure 5.3 further demonstrates the process. Task τ_1 and τ_3 both self-suspend themselves, and wait for the data from τ_2. The self-suspension interval $S_1(S_3)$ for task τ_1 (τ_3) depends on the execution of task τ_2, which in turn depends on task scheduling. We compute the worst-case task self-suspension interval next.

Theorem 5.1. Let us consider a task set τ on a uniprocessor consisting of periodic real-time tasks. In our task model, the maximum delay (represented by self-suspension interval S_i) for a job of τ_i due to shared resource accesses by jobs of other tasks $\tau_j \in \tau$ is given by [43]

$$S_i = \sum_{\tau_j \in \tau,\, j \neq i} \min(\mathrm{C}_j, B_j) + B_i, \tag{5.1}$$

where C_j denotes the worst-case length of the computational time of τ_j, and B_i and B_j denote the upper bound on retry loops for tasks τ_i and τ_j respectively.

Henceforth, we drop the t superscript from a task τ_i^{t} and generalize it with the following tuple: $(T_i, C_i^1, S_i, C_i^2, \upsilon_i, C_i^3)$, where non- trusted computation sections (C_i^q $q = 1, \ldots, 3$) are

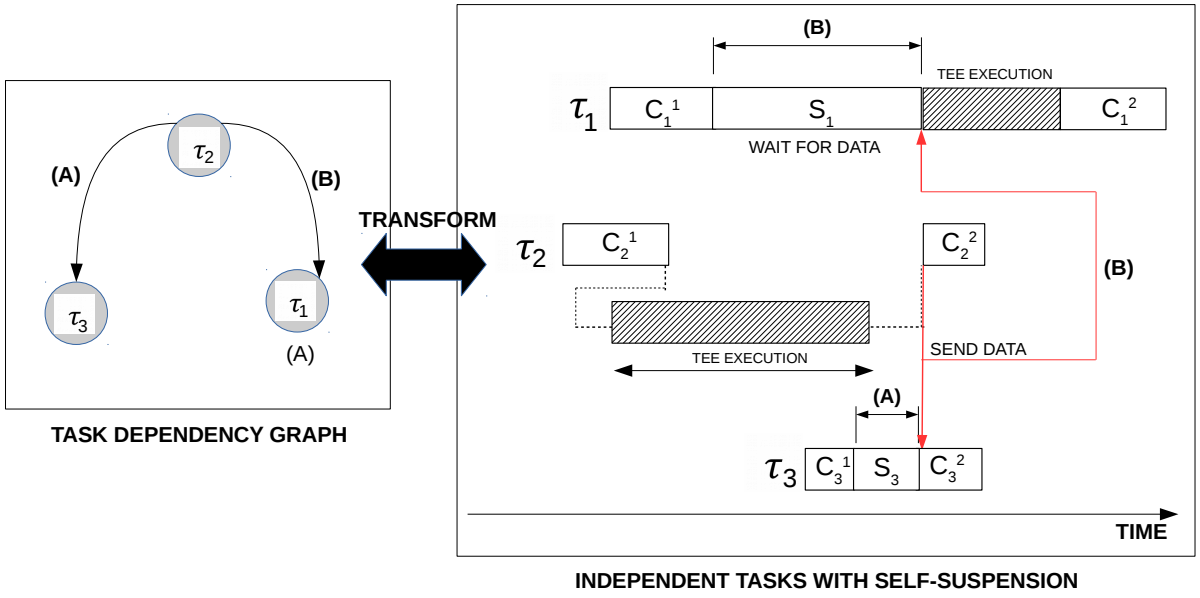


Figure 5.3: An example illustrating how task dependency and TEE execution are captured by our self-suspending TEE-task model.

interleaved with one self-suspension interval (S_i) to account for inter-task data dependency, and a TEE execution (υ_i) section. An ordinary real-time task (with no TEE requirements) will have $\upsilon_i = 0$. Similarly, a task with no inter-task data dependency can be represented by assigning $S_i = 0$.

5.3.2 Creating Trusted Execution Segments

A TEE section delimits the private and/or sensitive section of the source code (Figure 6.2). We list the process of partitioning a TEE-task into a secure and non-secure executable. Step 1 takes a task with trusted execution requirements, identifies the TEE section, and detaches it from the original source code. In step 2, we take the detached TEE segment, instrument it with TEE-specific internal APIs, and create a separate executable known as trusted application (TA). Similarly, in step 3, we instrument the non-TEE section of the code with TEE-specific interfaces to link it to the TA, and create a separate non-secure

executable. Finally, the two executable form a sequential flow of execution wherein the non-secure executable in the non-trusted OS starts its execution followed by a context switch to the TEE to run the TA in step 4. Ultimately, the TA execution returns the context back to the non-secure executable to complete further execution.

As an example, we select the PX4 module [3] sensors.c, locate its region of interest (ROI) which, we assume, requires TEE execution (shown in Figure 5.4) and draw a comparison with the steps explained in Figure 6.2. We take advantage of an existing work [85] to automatically partition the code around the ROI into a non-trusted executable shown as Partition 1 Figure 5.4 (equivalent to Figure 6.2 step 3), and a trusted application as Partition 2 of Figure 5.4 (equivalent to Figure 6.2 step 2).

5.4 Motivations & Problem Statement

TEE leverages hardware security extensions to provide platform virtualization to run a minimal secure kernel alongside a fully functional normal OS. The secure kernel extends a secure isolated abstraction from the rest of the system. While TEE provides industry-standard certification, security and isolation, its practical feasibility in hard real-time systems is questionable as there is a real-time specific challenge that needs to be overcome.

Namely, the timing overhead (Table 5.2) associated with secure monitor calls (SMC), described in Section 4.2.2, to setup and destroy a trusted execution environment is exceptionally high. This may adversely affects the timeliness of the system. Note that while it is in theory possible to account for such TEE-related overhead in a task's worst-case execution time, doing so would likely result in a gross under-utilization of resources. In addition, the overhead associated with TEE cannot be effectively amortized by dedicating specific core(s) to only TEE executions, as data communications among tasks must occur outside of TEE

```
1 Code snippet of original sensors.c module of PX4
2  float  air_temperature_celsius=0;
3  //Segment BEGIN: assume it requires TEE execution
4 air_temperature_celsius=(_diff_pres.temperature>-300..);
5 _airspeed.timestamp = _diff_pres.timestamp;
6  //Segment END
7  ...
8 _airspeed.air_temp_celsius=air_temperature_celsius;
9
10  Partition 1: Non-secure Code
11 Modified sensors.c module of PX4 to enable TEE exec
12 //wait to complete data dependency
13  float  air_temperature_celsius=0;
14  //Start TEE execution
15 TEEC_InitializeContext(NULL, &ctx); //set up TEE context
16 TEEC_OpenSession(&ctx,&sess,&uuid,..); //TEE entry
17  ...
18 TEEC_InvokeCommand(&sess,..., &op,..); //call trusted app (TA)
19 air_temperature_celsius=op.params[2].value.a; //store O/P
20 TEEC_CloseSession(&sess);
21 TEEC_FinalizeContext(&ctx);
22  //End TEE execution and return to normal execution
23 _airspeed.air_temp_celsius=air_temperature_celsius;
24
25  Partition 2: Trusted Application which runs in TEE
26 Code snippet of the corresponding partitioned TA
27  static  TEE_Result inc_value(uint32_t param_types,..)
28 { //Trusted computation
29 params[2].value.a=((*diff_buf).temperature>-300...);
30 return TEE_SUCCESS; //return TA O/P
```

Figure 5.4: Transforming original sensors.c module of PX4 to run in TEE.

Table 5.1: Example Task Set

Task (τ)	C^1	S	C^2	υ	C^3	D	T
τ_A	2	4	2	-	-	8	10
τ_B	1	-	-	5	1	8	10
τ_C	2	-	-	-	-	8	10

for security reasons. In other words, a new TEE session must be instantiated every time communications take place. In this work, we address this problem by judiciously scheduling TEE sections.

Let us consider a task set $\Psi = \{\tau_A, \tau_B, \tau_C\}$ (Table 5.1) where tasks τ_A and τ_C are ordinary real-time tasks while τ_B is a TEE-task. Also, tasks τ_A is dependent on τ_B while task τ_C is independent. Figure 5.5 captures execution windows of active job instances of each task in Ψ. From Figure 5.5(A), it is evident that the job instance of task τ_C misses its deadline on p_2 at time instant 7 even though p_1 remain idle due to self-suspension of a job of task τ_A (blocked to meet data dependency by a job of task τ_B) from time instant 2 to 6. However, in Figure 5.5(B), migrating the TEE execution section υ_B of task τ_B to p_1 allows job of task τ_C to meet its deadline.

We now identify a unique feature of TEE execution, which would justify the migration of TEE segments as described above. The migration cost associated with moving data to and from cores for TEE segments is usually much lower than that of an ordinary real-time task. This is due to the fact that a trusted execution usually requires small data sets and that TEEs typically provide limited features and functionality. Hence, it is preferable to migrate TEE sections instead of tasks that execute outside of a TEE. More importantly, from a security point-of-view, migration of TEE sections has no impact on security or protection. First, sensitive input data and code are only copied to a secure memory location in TEE after a core has switched to the secure mode. Second, each component of the memory subsystem consists

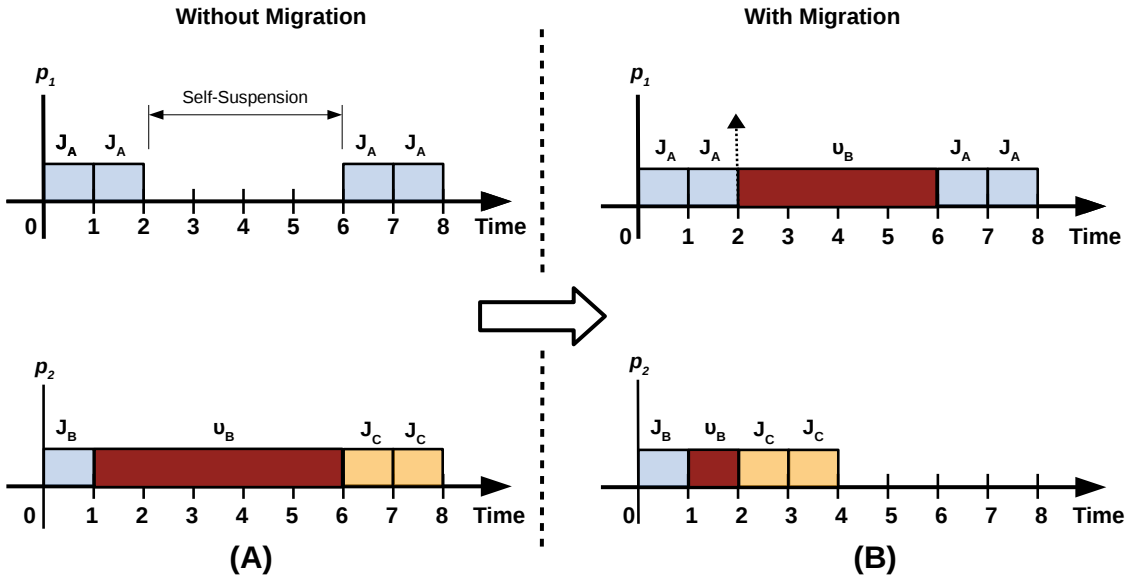


Figure 5.5: TEE section migration example. (Job J_B migrates its TEE section υ_B to core p_1, allowing J_C to begin execution earlier).

of segmented secure and non-secure blocks and TEE-related data is copied from encrypted external storage devices into a secure segment of the main memory and subsequently into a secure segment in the private cache. Similar steps are followed when writing outputs. Only TEE has access privilege to the data and instructions in secure hardware, thereby allowing sensitive portions of code to run in an isolated secure execution environment.

Problem Statement: Given a set of hard real-time tasks described in Section 5.3, and a set homogeneous cores P where each core $p_i \in$ P can only be operating either in the normal RTOS mode or in the TEE mode at a given time instant, our goal is to find an assignment and schedule such that all deadlines are met.

5.5 T-EDF: A Global Scheduling Algorithm

There exists several normal real-time task assignment and scheduling algorithms in literature [14, 15, 19, 35, 39, 42, 48]. Among them, global EDF (G-EDF) [16] is likely the most well-studied algorithm. In G-EDF, a single, global ready job queue is maintained in an earliest deadline first manner. In a homogeneous multicore system consisting of |P| cores, where P is the set of processor cores, the first |P| jobs at the head of the queue are dispatched for execution. In this work, the real-time task model presented in Section 5.3 is a special case of an existing self-suspending task model. While most research on self-suspending tasks focuses on fixed-priority uniprocessor scheduling [31, 63], a few consider multicore systems [22, 101]. G-EDF is also the state-of-the-art method for scheduling self-suspending tasks on multicore systems [83]. One drawback of G-EDF is that the number of migrated jobs may be large. Since migration cost can be significant [20], a restricted EDF (r-EDF) algorithm was proposed by Baruah et. al. [18]. In r-EDF, jobs are mapped to cores using the utilization based first-fit algorithm and each job, once assigned, does not migrate. Both G-EDF and r-EDF are oblivious to self suspension and while r-EDF provides a worst-case utilization bound comparable to, and under certain conditions better than, existing priority-driven partitioned scheduling algorithms, it fails to maximize the resource utilization of the system, such as in the case of G-EDF. To tackle these challenges, we propose a suspension-aware global scheduling algorithm, T-EDF, which not only, (i) improves the usable system utilization compared to both G-EDF and r-EDF but also, (ii) reduces the migration cost compared to G-EDF, as experimentally validated in Section 6.6.

One important observation is that the actual length of self-suspending intervals depend on the task schedule, as tasks are blocked until data is available. With the relatively low migration cost of a TEE-task (as discussed in Section 5.4), our goal is to modify r-EDF to

Table 5.2: TEE execution overhead

API name	Latency in us
TEEC_InitializeContext()	200
TEEC_OpenSession()	17000
TA_InvokeCommandEntryPoint()	250
TEEC_CloseSession()	1200
TEEC_FinalizeContext	100

only migrate trusted execution sections to an idle core to allow the next job to immediately execute on the original core, especially since migrating a TEE-task does not have an effect on the security of the system, as discussed in Section 5.4. Recall that trusted execution sections tend to be long, as shown in Table 5.2, and hence the potential finish times[2] of subsequent jobs on the original core are likely to be greatly reduced. As such, r-EDF can be considered a special case of our proposed algorithm, T-EDF. We now discuss T-EDF in detail.

5.5.1 Assigning Jobs to Cores

In r-EDF, the utilization-based first-fit policy is used to assign jobs to cores [18], which promotes predictability but does not make use of runtime information to optimize resource usage. In order to judiciously assign jobs to cores and facilitate later migration, we propose a method to calculate the degree of parallel suspension intervals (DOPS), which is a common suspension interval length that exists between any two jobs during their respective self-suspending interval. The upper bound on the suspension intervals (S_i for a task τ_i defined in Section 5.3.1) is used to calculate the DOPS. Our algorithm utilizes the value of DOPS to make a heuristic decision when performing job-to-core mapping.

[2]While it is possible that reducing the self-suspending intervals may impact sustainability, our results (Section6.6) suggest that the proposed approach and the worst case utilizable bound are safe. However, this issue will be the focus of future work.

Upon the arrival of a new job in the global ready queue, the DOPS values between this newly released job and all jobs that have been already assigned a core, are calculated. In doing so, we aim to improve core utilization without relying on a large number of job migrations. It is worth noting that while the actual degree of common suspension interval depends on the actual self-suspension intervals and preemptions, simulation results (Section 6.6) show that DOPS has good performance in general. To reduce runtime overhead, each core can keep track of the self-suspending time intervals as a job is assigned to it. Hence, this process takes constant time.

We now provide an example to demonstrate how DOPS can be computed. Figure 5.6 presents a snapshot of the system as it appears from a time epoch. There are three jobs ready to be assigned to cores, a job of τ_A (J_A), a job of τ_B (J_B) and a job of τ_C (J_C). In addition, jobs J_A, J_B, and J_C are released 10, 20 and 90 time units from the time epoch, respectively.

Algorithm 5.1 Degree of Parallel Suspension Intervals Calculation

▷ See below for description of parameters
procedure DOPS(t, ϵ, SS_r^s, SS_r^e, SS_i^s, SS_i^e)
    ▷ Parallel sleep starts between Jobs J_i and J_r
    $\mathrm{PSS} \leftarrow \max(\mathrm{SS}_r^s, \mathrm{SS}_i^s)$
    ▷ Parallel sleep ends between Jobs J_i and J_r
    $\mathrm{PSE} \leftarrow \min(\mathrm{SS}_r^e, \mathrm{SS}_i^e)$
    ▷ Overlapping parallel sleep time between Jobs J_i and J_r
    $\mathrm{DOPS} \leftarrow \mathrm{PSE} - \mathrm{PSS}$
    if $\mathrm{DOPS} \geq \epsilon$ then return Run task sequentially ▷ On same core
    else return Run task in parallel ▷ On different core

▷ t: current time, ϵ: user-defined threshold parameter, J_r: current instance of τ_r which has just been released, J_i: job already assigned to core
procedure Calculate_DOPS(t, ϵ, J_r, J_i)
    $\mathrm{SS}_r^s \leftarrow r_r + C_r^1 - t$ ▷ r_r is the release time of J_r
    $\mathrm{SS}_r^e \leftarrow r_r + C_r^1 + S_r^1 - t$
    $\mathrm{SS}_i^s \leftarrow r_i + C_i^1 - t$
    ▷ SS_i^s (SS_r^s) is offset time when self-suspension interval of J_i (J_r) starts, and takes value of $\emptyset$ if suspension interval has already begun
    $\mathrm{SS}_i^e \leftarrow r_i + C_i^1 + S_i^1 - t$
    ▷ SS_i^e (SS_r^e) is offset time when self-suspension interval of J_i (J_r) starts, and takes value of $\emptyset$ if suspension interval has already ended
    if $\mathrm{SS}_i^s \neq \emptyset$ and $\mathrm{SS}_i^e \neq \emptyset$ then
        return DOPS(t, ϵ, SS_r^s, SS_r^i, SS_i^s, SS_i^e)

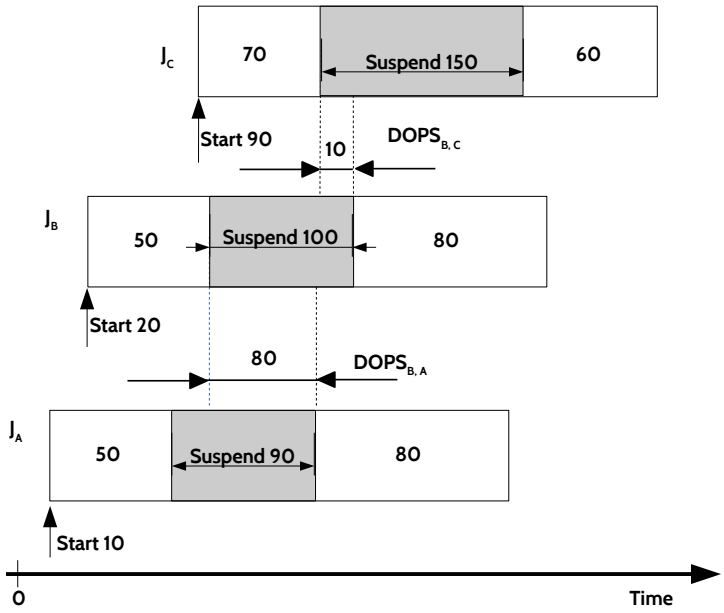


Figure 5.6: Example depicting the calculation of the degree of parallel suspension intervals (DOPS) for job J_B. Both $\text{DOPS}_{B,A}$ and $\text{DOPS}_{B,C}$ are calculated.

Their self-suspension interval lengths are 90, 100, 150 time units, respectively. The degree of parallel suspension interval (denoted as $\text{DOPS}_{B,A}$ between jobs J_B and J_A) is 80 time units. Similarly, $\text{DOPS}_{B,C}$ is 10 time units. If two jobs mutually have a high DOPS value, the probability that the trusted execution of the first job overlaps with a self-suspension time of the other job is low. This implies that, if the jobs are assigned to different cores, e.g., p_1, p_2 $\in$ P, it is unlikely that said TEE execution on p_1 will be able to migrate to p_2. On the other hand, a lower value of DOPS for any two jobs means that a successful migration of a TEE execution is more likely. We propose a threshold-based approach for calculating DOPS. An appropriate threshold value is a user-defined parameter and can be determined offline via profiling. Algorithm 5.1 describes how DOPS can be calculated, i.e., $\text{DOPS}_{x,y}$ for any two jobs J_x and J_y.

Once the DOPS of the newly released job is computed against each of the already assigned jobs to core, Algorithm 5.2 describes the process of assigning newly released jobs to core.

The main idea is that the next ready job is assigned to the core with the largest DOPS, i.e., we aim to maximize the probability that the jobs assigned to a given core run with as few idle time intervals as possible. As can be seen from Figure 5.6, job J_B will prefer to run in parallel with J_C and on the same core as J_A. The time overhead associated with DOPS calculation, and the subsequent dynamic job assignment policy, is analyzed in Section 5.6.

5.5.2 Migrating TEE-tasks

Once jobs have been assigned to cores, a local scheduler selects the job with an earliest deadline for execution, similar to r-EDF. Different from r-EDF, T-EDF permits migration of TEE executions to balance resource usage against migration overhead, as described in Algorithm 5.3.

Algorithm 5.2 Job to Core Assignment Algorithm

```
1: JL ← set of newly released jobs
2: procedure Assign(JL)
3:     for J_i ∈ JL do
4:         counter_j ← 0, j = 1, ..., |P|                ▷ P is the set of cores
5:         for p_j ∈ P do
6:             if p_j.ready = ∅ then
7:                 ▷ p_j.ready is the local runqueue for core p_j
8:                 p_j.ready ← J_i                        ▷ Assign job to core
9:             else
10:                for J_l ∈ p_j.ready do
11:                    ▷ Algorithm 1 for new job (J_i)
12:                    R ← CALCULATE_DOPS(t, ϵ, J_i, J_l)
13:                    if R == Run sequentially then
14:                        counter_j ← counter_j + 1
15:        index ← −1
16:        maxVal ← −1
17:        for p_j ∈ P do
18:            ▷ Find the highest voted core for job (J_i) assignment
19:            if counter_j > maxVal then
20:                index ← j
21:                maxVal ← counter_j
22:        ▷ Assign job (J_i) to core p_index
23:        p_index.ready ← p_index.ready ∪ J_i
```

When a job self-suspends, there are two possibilities. In the first case, another job already in the local ready queue starts executing. In the second case, the local ready job queue is empty. If another core is currently executing (or is about to execute) a TEE section, said TEE-task will be migrated onto the currently idle core. If two or more cores have TEE executions, ties are broken in favor of the TEE-task with an earlier start time. Note, as described earlier, that the migration cost associated with moving data to and from cores is usually much lower than that of an ordinary real-time task. Also, from the security point-of-view, migration of TEE-tasks has no impact on the security/protection (Section 5.4).

An example is given in Figure 5.5. Job J_A is running on core p_1 while J_B and J_C have been assigned to core p_2. J_B is a TEE-task. Job J_A self-suspends at time 2, resulting in an idle core p_1. Without migration, the TEE-task executes from time 1 to 6, at which point J_3 would start. Since p_1 is idle at time 1, the TEE-task can migrate to core p_1, allowing J_C to start at time 2. In other words, the migration of a TEE-task allows jobs that are waiting in the local ready queue to execute while unused idle time is reclaimed from another core. As for the core that the TEE-task migrates to (p_1 in our example), when a job becomes available, the migrated TEE-task (if it is still running) is moved back to its original core, thus ensuring that the execution of the TEE-task has a minimum impact on the other jobs.

5.6 Analysis of T-EDF

We analyze T-EDF in terms of time complexity and worst-case utilization bound in this section. Its performance, including the impact of our heuristic method to calculate DOPS, will be discussed in Section 6.6. The time complexity of T-EDF, the proposed global scheduling algorithm, is dominated by Algorithm 5.2, which requires $O(|\text{JL}| \cdot |\text{P}| \cdot \alpha)$, where JL is the ready job list at a given time instant, P is the set of cores in the system, and

Algorithm 5.3 Scheduler with TEE Section Migration

```
1: ▷ Some checkpoints are omitted to improve readability
2: procedure Scheduler(P)
3:     while true do
4:         if a new job J_i is released then
5:             JL ← J_i ∪ JL                                   ▷ Push job to JobList(JL)
6:             ASSIGN(JL)                                      ▷ Job to core assignment
7:         for p_j ∈ P do                                      ▷ For each core p_j
8:             if ¬ p_j.Active then                            ▷ If core is idle
9:                 ▷ Run the job (J_exe) with the earliest deadline
10:                J_exe ← p_j.ready[0]
11:                p_j.Active ← true
12:            if p_j.Active and J_exe.blocked and p_j.ready.size() < 2 then
13:                ▷ Core is idle since its current job (J_exe) is self-suspended
14:                for p_l ∈ P, l ≠ j do                       ▷ Find a core that is active
15:                    if p_l is executing a TEE section then
16:                        Migrate the TEE section to p_j
```

$\alpha = \max_{i=1,\ldots,\mathrm{P}} |p_i.\mathrm{ready}|$, $p_i \in \mathrm{P}$ is the longest local ready queue among all the cores at a given time. Hence, the time complexity of Algorithm 5.2 is $O(|\mathrm{JL}| \cdot |\mathrm{P}|)$.

Since the work in this paper targets hard real-time systems, we derive a worst-case utilization bound for T-EDF. Improving the utilization based schedulability bound, for example through a selective suspension-to-computation conversion (SSCC) approach [46][3], is left as future work. We leverage existing analysis for r-EDF [18] and EDF-based self-suspending tasks [43], and reproduce the two theorems below for ease of reference.

Theorem 5.2. A task set τ consisting of independent periodic real-time tasks must adhere to the following worst-case utilization bound to be feasibly scheduled by r-EDF on $|\mathrm{P}|$ processor cores [18].

$$U_{sum}(\tau) \leq |\mathrm{P}| - (|\mathrm{P}| - 1) \times U_{max}(\tau), \tag{5.2}$$

where

[3]SSCC selectively adds self-suspension intervals to certain jobs' computation times, thereby reducing the overall suspension induced CPU idle time.

$$U_{sum}(\tau) = \sum_{\tau_i \in \tau} \frac{C_i}{T_i}, \tag{5.3}$$

$$U_{max}(\tau) = \max_{\tau_i \in \tau} \left\{ \frac{C_i}{T_i} \right\}. \tag{5.4}$$

It is trivial to see that when no migrations are allowed, T-EDF reduces to r-EDF since both the algorithms use greedy approaches (DOPS and first-fit respectively) to improve system-wide resource utilization. Hence, Theorem 5.2 would also provide a worst-case utilization bound for T-EDF, except that r-EDF consider independent periodic real-time tasks. Since we have self-suspending tasks we leverage the maximum delay calculation (Theorem 5.1) due to self-suspensions (S_i) to calculate the utilization bound of T-EDF, as shown next.

Lemma 5.1. A task set τ consisting of self-suspending real-time tasks must adhere to the following worst-case utilization bound to be feasibly scheduled by T-EDF on |P| processor cores.

$$U'_{sum}(\tau) \leq |\mathrm{P}| - (|\mathrm{P}| - 1) \times U'_{max}(\tau), \tag{5.5}$$

where

$$U'_{sum}(\tau) = \sum_{\tau_i \in \tau} \frac{C_i + S_i}{T_i}, \tag{5.6}$$

$$U'_{max}(\tau) = \max_{\tau_i \in \tau} \left\{ \frac{C_i + S_i}{T_i} \right\}. \tag{5.7}$$

Proof. The lemma follows from Theorems 5.1 and 5.2 ■

5.7 Simulations

TEE is delimited by its secure memory footprint (Section 5.4). Since the maximum available private secure cache size in TEE is limited by its architecture-specific design constraint, the

task-specific secure cache footprint delimits the number of TEEs that can run simultaneously, thereby ensuring that TEEs do not exceed the secure cache limit. Similarly, our hardware testbed is limited by its available core count. Therefore, we extend the validation of our proposed approach (T-EDF) and assess its real-time performance against global EDF (G-EDF) [16] and restricted-migration EDF (r-EDF) [18] in a simulated environment using randomly generated task sets. We compare the performance of T-EDF with both r-EDF and G-EDF as T-EDF is based on r-EDF. In addition, since T-EDF allows migration, we compare and contrast it against G-EDF.

Table 5.3 represents 3 different variants of task sets that are used to compare T-EDF against the existing G-EDF and r-EDF. For each utilization level (100%, 150%, 200%, 250%,. . ., 400%), we generated 100 task sets of 10 tasks each. For use case scenario #1, each task set comprises of average number of TEE tasks (50% of the tasks in the task set). The trusted execution duration (υ_i) of each TEE task is set at 30% of the worst-case execution time. The self-suspension interval (to account for inter-task data dependency) is also set at 30% of the worst-case execution time. Similarly, for use case scenario #3, each task set comprises of high number of TEE tasks (90% of the tasks in the task set). The trusted execution duration (υ_i) of each TEE task is set at 70% of the worst-case execution time. The self-suspension interval is also set at 70% of the worst-case execution time. We consider a system consisting of 4 processor cores in total. Each simulation is carried out for one hyperperiod, the least common multiple of the periods of all the tasks in a task set. We also test the scalability of our approach by varying the core count $(2, 3, 4, \ldots, 10)$.

We now assess the performance of our proposed job assignment and scheduling algorithm. As previously stated, T-EDF aims to tackle three challenges: (i) improve schedulability, (ii) reduce migration overhead, and, (iii) effectively utilize idle processor cycles. Figures 5.7(a)-

[3]Read as Item[small/short-avg-high/long)]: #TEE tasks[30%−60%−90%], TEE length[30%−50%−70%], Self-suspension length[30% − 50% − 70%]

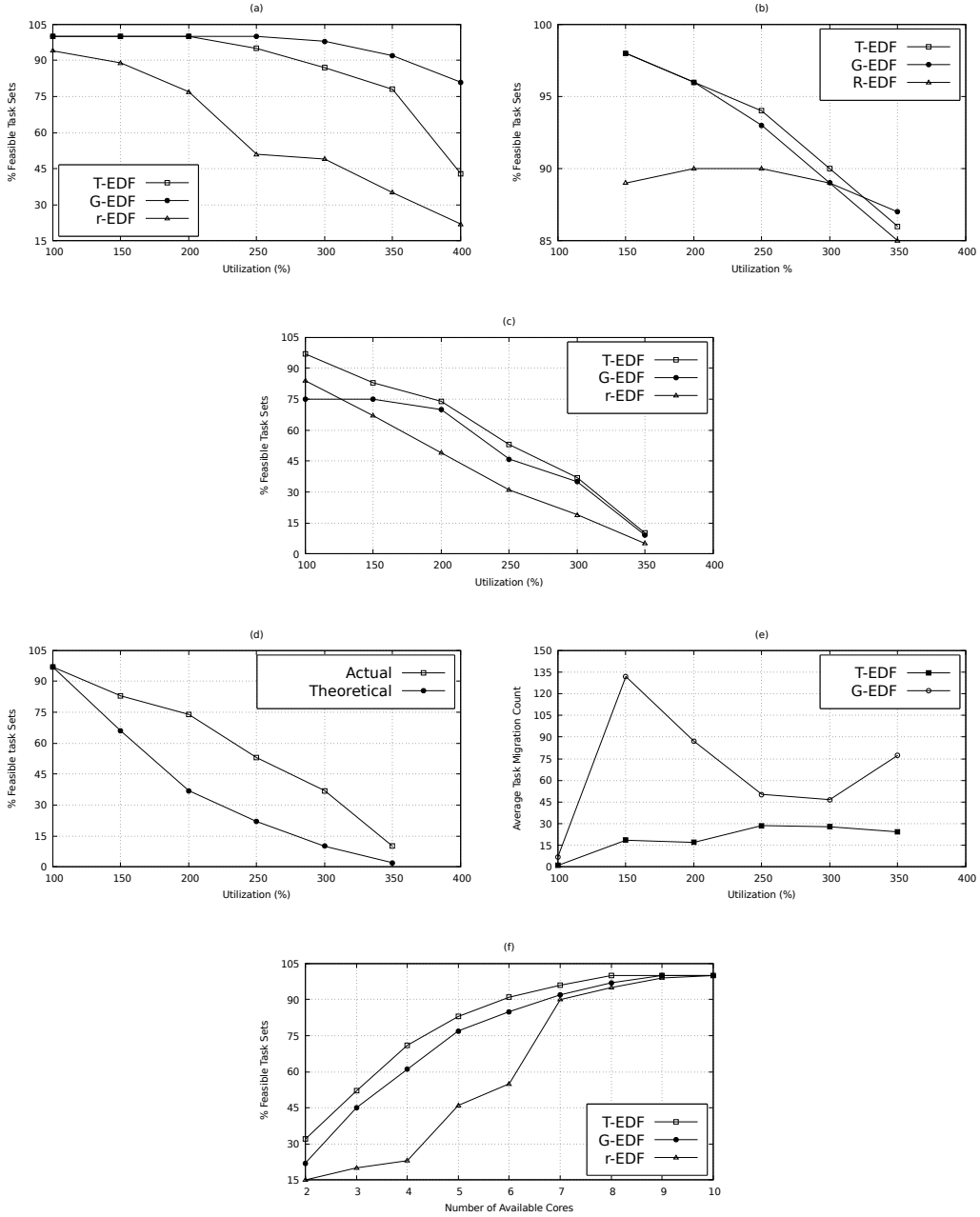


Figure 5.7: Simulation results showing (a), the percentage of feasible task sets as a function of utilization for use case scenario #1 (Table 5.3) (b), the percentage of feasible task sets as a function of utilization for use case scenario #2 (Table 5.3) (c), the percentage of feasible task sets as a function of utilization for use case scenario #3 (Table 5.3) (d), the percentage of feasible task sets of T-EDF as a function of utilization obtained by simulation and feasibility test (use case scenario #3) (e), average task migration count as a function of utilization for T-EDF and G-EDF (use case scenario #3), and (f), the percentage of feasible task sets as a function of core count (use case scenario #3).

Table 5.3: Simulation use case scenarios

#	# TEE tasks	TEE length (v_i)	Self-suspension length(S_i)
1	avg	short	short
2	avg	short	long
3	high	long	long

5.7(f) compare the simulation results to highlight the benefits of T-EDF over G-EDF and r-EDF. Note that both G-EDF and r-EDF are suspension oblivious algorithms. Therefore, we add the self-suspension intervals as part of a task's worst-case execution time for G-EDF and r-EDF.

Figure 5.7(a) reports the simulation results comparing the performance of T-EDF against both G-EDF and r-EDF in terms of the percentage of feasible task sets as a function of utilization demands for use case scenario #1 (Table 5.3). The trend shows that G-EDF improves the usable utilization bound by 9% and 26% on average over T-EDF and r-EDF, respectively, across all utilization levels. T-EDF fails to improve the usable utilization bound set by G-EDF because the tasks do not have enough inter-task data dependency (attributed by short self-suspension intervals) for TEE migrations to exploit optimal resource utilization in T-EDF. However, with longer self-suspension intervals as in use case scenario #2 (Table 5.3), Figure 5.7(b) shows that T-EDF improves the usable utilization bound by 5% on average over r-EDF, while performing on par with G-EDF across all utilization levels. Interestingly, this is the best case scenario for all the scheduling algorithms.

For the next set of results we will be using simulation use case scenario #3 (Table 5.3). Figure 5.7(c) reports the simulation results comparing the performance of T-EDF against both G-EDF and r-EDF in terms of the percentage of feasible task sets as a function of utilization demands. The trend shows improved usable utilization bound of 29% and 8% on average over r-EDF and G-EDF, respectively, across all utilization levels. Note that all

algorithms policies fail to feasibly schedule task sets once the task utilization demand exceeds 350% utilization.

Figure 5.7(d) compares the average performance T-EDF against the worst-case theoretical bounds (derived in Section 5.6) and confirms its correctness through simulation results. We compare the worst-case percentage of feasible task sets (using the utilization bound) with the actual percentage of feasibly scheduled task sets in simulation. Our results indicate a sufficient and conservative utilization bound which can seamlessly be used to schedule hard real-time task sets. Figure 5.7(e) reports the simulation results comparing the performance of T-EDF against G-EDF in terms of the number of migrations. The trend shows an improvement of 26% on average and up to 86% across all utilization levels. This can be attributed to the design policy of T-EDF which prohibits a task migration to another core unless it is a TEE-task. This is in direct contrast to G-EDF which allows unbounded task migrations. Since r-EDF does not allow any task migration, we skip the comparison between our solution and r-EDF. Figure 5.7(f) reports the simulation results comparing the scalability of our approach. It compares the performance of T-EDF against both G-EDF and r-EDF in terms of percentage of feasible task sets as a function of number of available cores. We report an average improvement of up to 10% and 50% over G-EDF and r-EDF respectively across all utilization levels.

For our simulation results, we identify the worst-case scenario for our proposed solution when a task set consists of tasks (1) which have minimum or no suspension interval, (2) has minimum or no TEE tasks, and, (3) has shortest WCET of trusted execution sections in TEE tasks. This complies with a task set which is similar to a normal real-time task set, consisting of tasks without TEE execution requirement. Such a task set may be scheduled feasibly with the existing r-EDF and G-EDF. However, since our proposed scheduling framework leverages r-EDF, the probability of obtaining a feasible schedule for a task set by our proposed

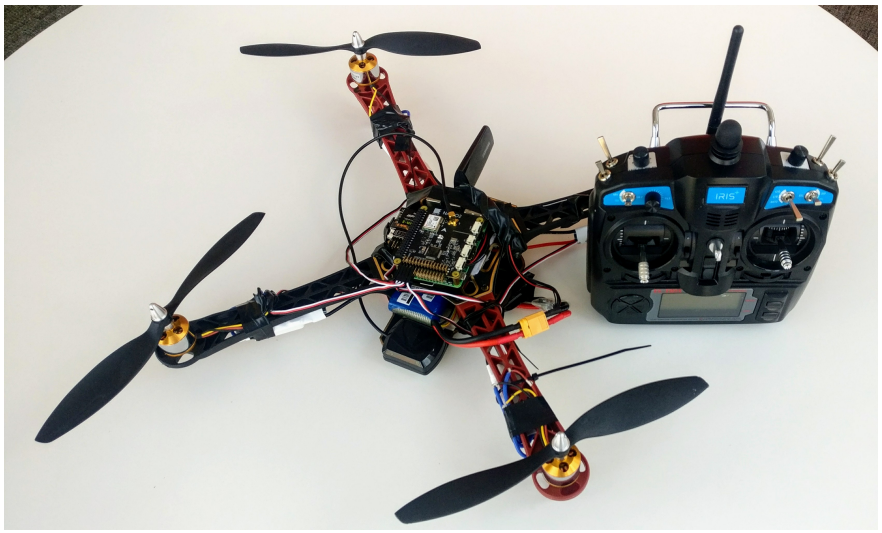
framework is exactly the same as the usable utilization improvement we have reported over r-EDF.

5.8 Experiments

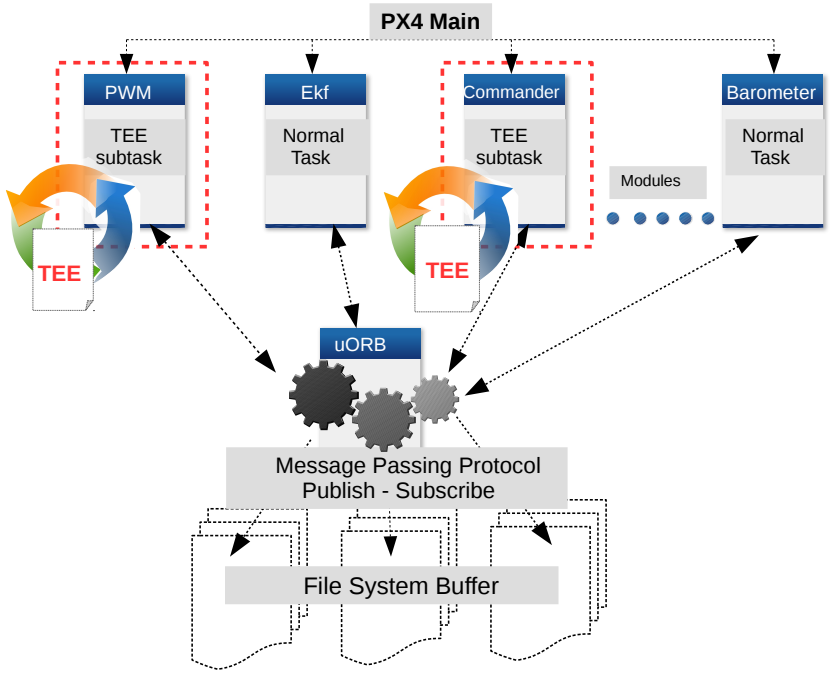
To validate our proposed approach in a realistic setting, we selected quadcopters as a representative COTS-based mission-critical hard real-time system for a variety of reasons. First, quadcopters are currently the most popular type of unmanned aerial vehicles (UAVs) [27] and a widely used variant of drones. Secondly, quadcopters may contain sensitive TEE-specific code such as coordinates for surveillance, tracking data and algorithms for specific targets on the ground, as well as real-time requirements. Thirdly, quadcopters are easier to deploy and exhibit simpler hovering capabilities and flight dynamics as compared to other types of autonomous aerial vehicles such as helicopters or airplanes, making them suitable for a proof of concept.

5.8.1 Experimental Setup

A picture of our testbed is shown in Figure 5.8(a). The Raspberry Pi 3B (Rpi 3B) is a small computer powered by Broadcom BCM2837 chipset consisting of quad-core ARM Cortex A53 processor, 1 GB LPDDR2 RAM, and numerous sensors (for eg., Barometer, IMU etc.). It can run Linux and other non-trusted operating systems. It also extends support for ARM TrustZone. The Navio2 HAT (Hardware Attached on Top) adds two inertial measurement units (IMUs) and an 8-channel PWM output via a separate micro-controller to communicate with the Raspberry Pi. We used the Linux RT_PREEMPT Kernel v4.6.3 to build our prototype T-EDF by modifying the Linux real-time scheduling class SCHED_DEADLINE.



(a)



(b)

Figure 5.8: (a) A custom-made quadcopter used as a hardware testbed and (b) PX4 flight stack workflow depicting inter-task dependencies according to a message passing protocol.

We run the modified real-time Linux OS (as non-trusted OS) along with OP-TEE OS [2] (as TEE) on the Rpi 3B that extends the support for ARM v8 embedded virtualization.

While there are several open-source autopilot software stacks, we focus on PX4 [3], a Linux compatible open-source project with real-time properties. PX4 contains several modules, each of which is considered a task in our context and whose function is briefly described in Table 5.4. The modules spawn priority-bound stand-alone periodic thread(s). Each iteration of these threads consist of three distinct phases:

1. Acquire flight status through external sensor data.
2. Use software controller to prescribe corrective measures.
3. Implement the changes through external actuators.

The multi-threaded flight stack follows a publish-subscribe communication protocol [49] with resource sharing and inter-process communication. Figure 5.8(b) shows a detailed realization of the publish-subscribe model of the PX4 flight stack. A central module UORB acts as the front-end of a shared buffer which is tasked with data storage. All the other modules communicate with UORB though a publish-subscribe message passing protocol. For example, the Barometer module acquires the in-flight atmospheric pressure through external sensor data. It stores the data into the shared message buffer. The Ekf module uses a software controller logic to prescribe a corrective measure if necessary and publishes it through the message passing protocol. The PWM module subscribes to the published data and implements the prescribed changes through external actuators.

PX4 is a multi-threaded real-world application which uses FIFO scheduling by default. While we implemented T-EDF on the testbed, the open-source port of TEE (OP-TEE) does not have provision for most C++ core libraries nor support certain operations, e.g., floating point

Table 5.4: PX4 Module List (Higher value means higher priority)

Module Name	Priority	Description
PWM Output	99	Motor control
Land Detector	98	Takeoff and landing state detector
Barometer	98	Hardware wrapper for barometer
ekf2	95	Kalman filter based stabilizer
sensors	94	Sensor data fusion and publish
position control	94	Position estimation and control
attitude control	94	Attitude estimation and control
commander	89	UI for takeoff, landing etc.
navigator	54	ground control-quad link

calculations. Similarly, to schedule PX4 using T-EDF (even with SCHED_DEADLINE), significant source code modifications are required to OP-TEE and software stack. Since we do not wish to compromise the integrity of OP-TEE by making modifications to secure OS, we validated our implementation by creating a macrobenchmark, which mimics the execution flow and timing characteristics of PX4 flight stack. Certain modules are augmented with calls to TEE for secure execution. The code snippet of our representative macrobenchmark is shown in Figure 5.9.

5.8.2 Results

Table 5.5 presents the experimental results of the PX4 macrobenchmark on our hardware testbed (RPi 3B). For each task set utilization level (100%, 150%, 200%, 250%,..., 300%), we generated 100 task sets of 5 tasks each. The period (T_i) and worst-case execution time (C_i) of each task is randomly generated so long as the overall utilization of the task set remains within the corresponding utilization level. The worst-case execution time of a task τ_i is upper bounded by $\sum_{q=1}^{3} C_i^q + \upsilon_i$, where C_i^q denotes each non-secure execution segments, and υ_i denotes the trusted execution interval. Each task set comprises of 90% TEE tasks

Table 5.5: Summary of experimental results of PX4 macrobenchmark (Figure 5.9) running on Rpi 3B using T-EDF and G-EDF with respect to percentage of feasible task sets and average task migration count as a function of task set utilization.

Util (%)	T-EDF		G-EDF	
	Feasible task set (%)	Migration count	Feasible task set (%)	Migration count
100	94	0	74	3
150	83	8	71	128
200	72	8	66	97
250	53	31	46	54
300	37	29	35	52

and 10% ordinary real-time tasks. The trusted execution duration (υ_i) of each TEE task is set at 70% of the worst-case execution time. The self-suspension interval is also set at 70% of the worst-case execution time. We use all the 4 processor cores of Rpi 3B to run our experiments. Each experiment is carried out for one hyperperiod, the least common multiple of the periods of all the tasks in a task set. The trend shows improved usable utilization bound by 9%, and reduced migration count by 52%, on average over G-EDF across all utilization levels. As stated in Section 5.6, the worst-case computation overhead for DOPS calculation (for subsequent task assignment to a suitable core) is dependent on the the length of the ready queue. However, for the given task set parameters (Section 5.8.2), our algorithm utilizes less than 1% of the scheduler tick (10 μs in our case). In section 6.6, we see a constant improvement with an increase in the number of tasks in the task set.

Example

We run our example benchmark (Figure 5.9) on preempt-RT Linux Kernel v4.6.3 with our implementation specific modifications. It consists of three modules: Thread *inc_x* is an independent normal SCHED_DEADLINE task. We partition a TEE-task (see Section 5.3.2)

```
1 #define SCHED_DEADLINE 6 //Modified SCHED_DEADLINE (T−EDF)
2 void *inc_x(void *x_void_ptr){ //Normal deadline task body
3  ...
4 attrX.sched_policy= SCHED_DEADLINE;
5 ret = sched_setattr(0, &attrX, flags); //set EDF params
6  ...
7 while(++(*x_ptr) < 100); //normal computation
8  ...
9 return NULL;}
10 void *inc_y(void *y_void_ptr){//TEE task body
11 usleep (...) ; //simulate self−suspension for data dependency
12  ...
13 }
14 void *inc_z(void *z_void_ptr){//Trusted executable body
15  //Start TEE execution
16 TEEC_InitializeContext(NULL, &ctx);
17 TEEC_OpenSession(&ctx,&sess,&uuid,..);
18  ...
19 TEEC_InvokeCommand(&sess,..., &op,..);
20 TEEC_CloseSession(&sess);
21 TEEC_FinalizeContext(&ctx);
22  //End TEE execution
23  ...}
24 int main(){ //Main CFS task to spawn other SCHED_DEADLINE tasks
25 syscall(288, 1, 1, 0); //independent EDF task to run on core #1
26 pthread_create(&inc_x_thread, NULL, inc_x, &x);
27 syscall(288, 2, 1, 0); //TEE task to run on core #2
28 pthread_create(&inc_y_thread, NULL, inc_y, &x);
29 while(!OPTEE_sec_go){}//wait for TEE execution
30 syscall(288, 0, 1, 1); //TEE section to run on core #0
31 pthread_create(&inc_z_thread, NULL, inc_z, &z);
32 //wait for tasks to end
33 return 0; }
```

Figure 5.9: Code snippet of our representative macrobenchmark that mimics PX4 modules.

into a thread *inc_y* (another SCHED_DEADLINE task with a non-secure section of code) and *inc_z* which contains the code that needs to be runs in a TEE. Threads *inc_y* and *inc_z* realizes inter-task temporal data dependency by including an INTERRUPTABLE sleep duration (self-suspension interval). Each thread starts as a CFS task, and eventually switches to a SCHED_DEADLINE task. Each module is distinguished from the other by the scheduler through a bit combination realized through a tuple (cpu, OPTEE_on, OPTEE_task) and communicated through our custom built syscall to the kernel. The cpu denotes the assigned core (calculated using DOPS shown in Section 5.5.1), OPTEE_on categorizes T-EDF as the scheduling policy for the task and the OPTEE_task bit distinguishes a TEE-task from its designated trusted application. Note, that T-EDF prohibits task migrations with the exception of TEE-tasks. Hence, when the OPTEE_task bit is off, we set the scheduler flag NR_CPUS_ALLOWED to #1. This informs the scheduler that the current task will never be up for migration.

5.9 Summary

We present a scheduling framework that leverages the self-suspending task model to capture inter-task communication requirements of real-time applications which require TEE execution. Our solution includes a task assignment and scheduling algorithm that opportunistically steals CPU cycles to execute lengthy trusted task executions while maintaining hard real-time deadlines. The proposed approach was validated on a hardware testbed which reveal that our proposed algorithm outperforms existing approaches. Simulation results on synthetic benchmarks validate that the result is consistent with the theoretical bounds presented in the paper.

Chapter 6

Optimized Trusted Execution for Hard Real-Time Applications

Trusted execution environments (TEE) [114] are a widely used solution for industry standard platform-level security in embedded systems. TEE provides virtualization for a secure execution environment by leveraging architecture-specific hardware security extensions to protect and isolate information from access by a third party [50, 106]. The wide spread availability of TEE in commercial-off-the-shelf (COTS) systems (1) dramatically reduces the need for the costly development and manufacturing of custom hardware solutions such as hardware-based encryption, code obfuscation, etc. [28, 38], and, (2) allows for quick redeployment as platform virtualization on top of ARM TrustZone enables security extensions.

However, it may be challenging to adopt TEE in hard real-time systems, as TEE can incur large time overhead and highly variable execution times [85]. Specifically, each instance of TEE execution (in ARM TrustZone) is initiated by a setup phase and exits through a destroy phase. Since TEE leverages architecture-specific secure monitor calls (SMC) to realize these phases, the time overhead associated with TEE execution, including setting

up and tearing down a TEE session, requires 18,500 μs on a Raspberry Pi 3B (Rpi 3B) (Table 5.2). Moreover, data/instruction fetches and writebacks during TEE execution is a major cause for large variation in task execution times, weakening predictability.

6.1 Contributions

In this work, we focus on TEE supported by ARM TrustZone, as the latter is popular in smartphones and other embedded mobile devices [95], while Intel SGX primarily targets resource-rich servers [102]. Our objective is to reduce the overall number of SMCs in applications where multiple sections of the code that must run in TEE are fused together to (a) amortize the time overhead associated with SMCs, (b) minimize the associated I/O traffic (memory fetches and writebacks) since they are the primary source of variable time overhead, and, (c) improve the temporal predictability of the system by reducing the number of switching between secure and non-secure environments due to SMCs. We also present a task assignment and scheduling framework for real-time trusted execution on readily available multicore systems. Since our work focuses on optimizing trusted executions for use in hard real-time systems without changing the underlying TEE implementation, we do not examine the security benefits and drawbacks of TEE. Instead, readers are referred to existing work [56, 62, 130]. Our main contributions are:

1. We present our approach for reducing TEE overhead by forming super-TEEs, which involve fusing together two or more secure code or application sections that require TEE. We describe a task model to represent a super-TEE, discuss a technique to enforce timing correctness, and derive a sufficient condition for schedulability for super-TEEs and other real-time tasks on uniprocessor systems under a fixed priority scheduling scheme.

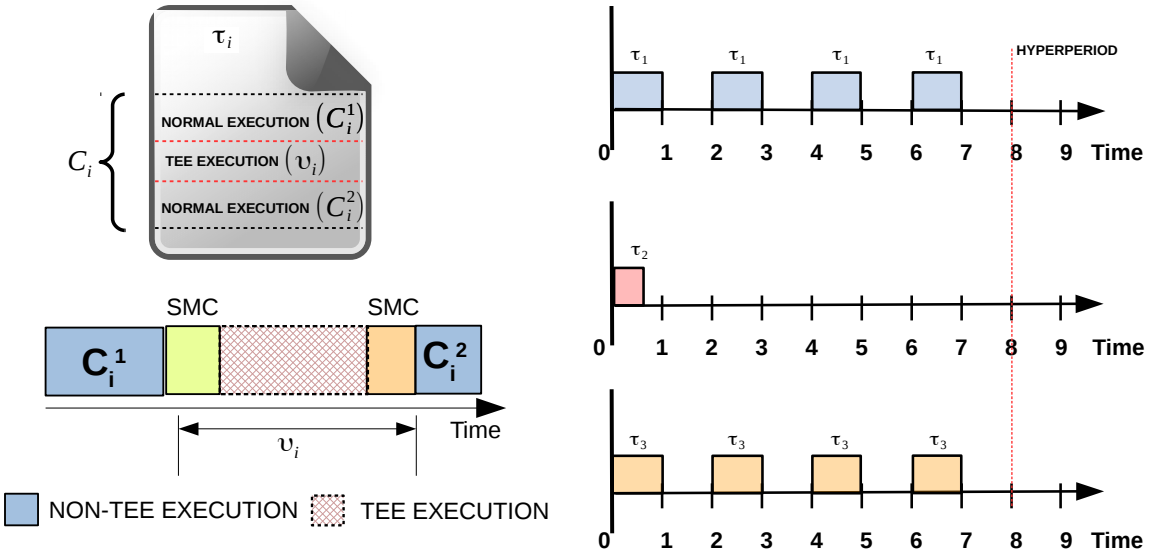
2. We propose a fixed-priority task-to-core assignment and partitioned scheduling technique called ct-RM for multicore systems. We show that our approach never performs worse, and in fact often outperforms the widely used first-fit partitioned rate-monotonic scheduling algorithm for multicore systems. We validate our approach and assess its performance in a custom-built simulated environment.

3. We validate our approach on a Raspberry Pi running Linux (with preempt-RT patches for real-time behavior) and OP-TEE, an open-source TEE implementation.

6.1.1 System Model

We assume a set of homogeneous cores P, where each core $p_j \in$ P, can switch between secure (TEE) and non-secure (non-trusted OS) mode of operation using SMCs. While the non-secure mode is used for executing normal real-time tasks, the secure mode runs TEE sections of code. In addition, we consider a set of n independent synchronous periodic hard real-time tasks, which have been re-factored to isolate TEE sections from its non-secure execution segments [85]. Therefore, each task τ_i, $i = 1, \ldots, n$, can be described by the following tuple: $(T_i, C_i^1, \upsilon_i, C_i^2)$, where C_i^1 and C_i^2 are the worst-case execution times (WCETs) in non-secure mode, and υ_i is the WCET in TEE mode, and T_i is the period of execution. We assume that deadlines are equal to periods. Figure 6.1(a) depicts our real-time task model. For a task with TEE requirements, two normal computation segments (C_i^q $q = 1, 2$) are interleaved by a single trusted execution segment[1] (υ_i). Similarly, for a task τ_i that does not require TEE, $\upsilon_i = C_i^2 = 0$. The upper bound on the combined WCET of a task τ_i is given by $C_i = \sum_{q=1}^{2} C_i^q + \upsilon_i$.

[1]Though we consider a single trusted execution interval for simplicity, it is not advisable to have multiple TEE execution sections interspersed with non-secure computation segments within a TEE task, since each trusted execution has high time overhead (Table 5.2).



(a) (b)

Figure 6.1: (a) Our real-time task model. For a task τ_i without TEE requirement, $\upsilon_i = C_i^2 = 0$; (b) Example offline profiling of the task set (Table 6.6) for super-TEE construction.

We consider partitioned scheduling in this work and will leverage the first-fit rate-monotonic (RM-FF) scheduling policy (Section 6.4). A sufficient utilization based test for RM for a uniprocessor is restated below. The utilization U_i of task τ_i is defined as $\frac{C_i}{T_i}$, and the utilization of a task set (U_T) is the sum of the utilization of all the tasks, n, in the task set, i.e., $\sum_{i=1}^{n} U_i$. Since we target multicore systems in this work, the Liu-layland (LL) limit [82], given by Equation 6.1, is used to perform per-core schedulability test when task to core assignment is carried out.

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1). \tag{6.1}$$

6.1.2 Creating Trusted Execution Segments

A TEE section delimits the private and/or sensitive section of the source code (Figure 6.2). We list the process of partitioning a task with TEE requirements into a secure and non-

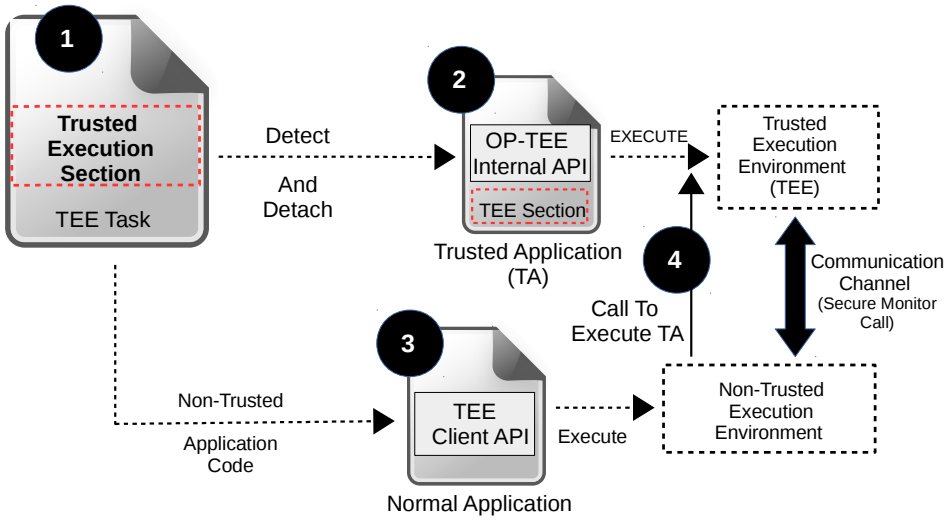


Figure 6.2: Step-by-step example showing how to convert a software program into separate executable.

secure executable. We use an existing work [85] to automatically partition the original code around a segment which is annotated by the software developer to demarcate for TEE computation into a non-trusted executable (step 3), and a trusted application (step 2) as shown in Figure 6.2. Step 1 takes a task τ_i with trusted execution requirements, identifies the annotated TEE section (υ_i), and detaches it from the original source code. In step 2, we take the detached TEE segment (υ_i), instrument it with TEE-specific internal APIs, and create a separate executable known as trusted application (TA). Similarly, in step 3, we instrument the non-trusted section(s) of the code (C_i^q $q = 1, 2$) with TEE-specific interfaces to link it to the TA, and create a separate normal non-secure executable. Finally, the two executable form a sequential flow of execution wherein the non-secure executable in the non-trusted OS starts its execution followed by a switch to the TEE to run the TA in step 4. Ultimately, the TA execution returns the context back to the non-secure executable to complete further execution.

6.2 Motivation

While TEE provides industry standard certification, security and isolation, its use in hard real-time systems is challenging. As previously reported [85], and verified through experimentation (Table 5.2), the time overhead associated with SMCs is large. The increase in WCETs for TEE execution can be primarily[2] attributed to architecture-specific SMCs to setup and destroy a trusted environment for secure execution. Note that while this increase in tasks' WCETs can be considered during schedulability analysis, many systems are resource constrained. In addition, it is not feasible to always keep TEE sessions open since a single TEE session is limited by the size of its secure private cache (4KB for our board). Said cache is used to store and execute the trusted executable, along with any other required data. Hence, the number of applications/trusted segments that can execute in a single session is limited. Even if the size of the secure private cache is large, always keeping TEE sessions open would mean dedicating a fixed number of cores to running TEE, resulting in a less efficient use of computing resources since these cores may often be idle. In addition, no intra-task communications are permitted during TEE execution by virtue of TEE security. For trusted segments requiring data for trusted computation, a new TEE session has to be open to load such data into the secure private cache, even if such inputs were generated from other trusted segments. In this work, we propose to amortize the time overhead associated with each call for secure execution in TEE by grouping two or more tasks which have TEE sections and fuse their trusted execution sections together into a super-TEE (Section 6.3).

We now provide a simple example to show how forming a super-TEE can help to decrease tasks' collective worst-case execution time (WCET). In this example we will ignore task

[2]Running a TEE section on a processor requires an exclusive access for the trusted OS to execute the trusted segment on that particular core (i.e., the core must be in secure mode and not normal mode). While interrupts exist to switch back to the non-trusted environment and service normal execution, a mode change adds to the time overhead associated with trusted execution.

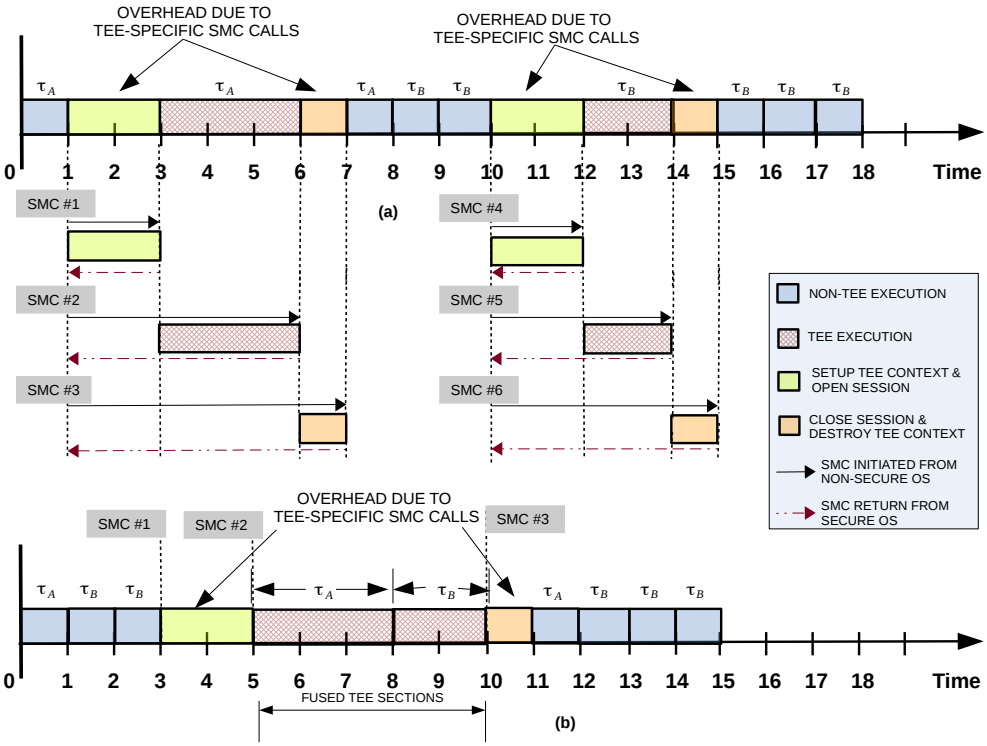


Figure 6.3: A motivating example where (a) two tasks need six SMC calls to access two separate instances of TEE execution, but, (b) fused TEE execution sections reduce the number of SMC calls to three.

Table 6.1: Task Set 1

Task (τ)	C_i^1	v_i^s	v_i^t	v_i^d	$v_i = v_i^s + v_i^t + v_i^d$	C_i^2	$C = \sum_{q=1}^{2} C_i^q + v_i$	T
τ_A	1	2	3	1	6	1	8	16
τ_B	2	2	2	1	5	3	10	17

deadlines for clarity. Consider two tasks τ_A and τ_B shown in Table 6.1. Let us assume that both tasks contain sections that require TEE execution. The upper bound on the non-trusted segment for both tasks τ_A and τ_B, given by $\sum_{q=1}^{2} C_i^q$, is equal to 2 and 5 respectively. Similarly, the upper bound on the trusted execution section, given by $\upsilon_i = \upsilon_i^{\mathrm{s}} + \upsilon_i^{\mathrm{t}} + \upsilon_i^{\mathrm{d}}$, is equal to 6 and 5 respectively, where SMC_setup is denoted by υ_i^s, SMC_destroy by υ_i^d and the actual trusted computation duration inside TEE by υ_i^t. Hence, C_A and C_B equal 8 and 10 time units, respectively. Figure 6.3(a) shows a possible schedule if we were to execute τ_A and τ_B separately. Figure 6.3(b) shows how the collective WCET of τ_A and τ_B, now 15 time units since $\upsilon_{AB} = 2 + 1 + 3 + 2 = 8$, is reduced when a super-TEE is formed. Clearly, reducing WCETs can help to improve schedulability. However, while fusing the TEE execution sections of tasks results in improved collective WCET, such a modification may change the collective period of the tasks under consideration, thereby, potentially violating the timing requirements of the tasks, and increasing the total system utilization. We address these challenges next.

6.3 Offline Super-TEE construction

We construct super-TEEs to amortize TEE execution overhead while maintaining logical and timing correctness. A super-TEE is defined as a real-time task consisting of a single secure section or code that require TEE execution, formed by fusing together the trusted execution sections of two or more real-time tasks. Since secure memory locations are allocated contiguously in TEE [2], we exploit the high spatial locality for secure memory locations to amortize the overhead associated with I/O transactions during trusted execution. We first introduce super-TEE and its construction while maintaining its logical correctness, followed by its representation using a real-time task model, and discuss its timing correctness. Fi-

nally, we present a technique to select which tasks to form super-TEEs to maximize the schedulability of the system.

6.3.1 Task Partitioning and TEE Fusions

The process of creating super-TEEs is carried out once offline. In this section, we discuss how to fuse TEE sections together. The question on which actual TEE sections to fuse together will be addressed in section 6.3.5. For the sake of simplicity, we consider fusing the TEE sections of two tasks in this paper. However, our approach can be applied to an arbitrary number of tasks with harmonic periods [94]. Section 6.1.2 explains the process of partitioning tasks with TEE requirements into separate executables. The υ_i section forms a separate trusted executable (TA) from the C_i^q $q = 1, 2$ sections which is collectively denoted by a client application (CA). The TA is invoked by the CA through SMCs (Figure 6.2).

We focus on independent tasks for super-TEE construction in this work. Before fusing the TEE sections, each task τ_i is profiled for its real-time periodic behavior. That is, each task is run in isolation on a uniprocessor to determine when it would execute. We then collect this information over a hyperperiod (Figure 6.1(b)). Then, an existing framework [112] is leveraged to calculate the overlapping execution sections for each task tuple, and the data collected is stored as entries in Table 6.7. We then ascertain whether the TA is independent by checking the memory page access (using an existing automated tool [85]) and record the memory footprint of each TA. Since the maximum available private secure cache size in TEE is limited by its architecture-specific design constraint, the task-specific secure cache footprint delimits the number of TEEs that can be fused together so that super-TEE does not exceed the secure cache limit. We merge both TA-specific code/data into a single executable and augment the fused TA to pre-fetch the memory address blocks of the TAs of the

fused tasks (example below). Merging multiple TEE sections into a single TA allows it to be loaded once, followed by multiple function calls, one for each of the TEE sections resulting in sequential execution. The trusted OS maintains the context of the secure execution (including instructions and meta-data not visible to the non-trusted OS). Our approach avoids the intermediate steps of unpredictable and time-consuming I/O transactions (memory writeback for the current TEE section of a task carried out while performing memory fetches for the subsequent TEE execution of the other task), leading to an improved overall temporal predictability. We also merge the CAs into a single executable while maintaining the overall temporal correctness of the tasks (see Section 6.3.3). To regulate our assumption, we ensure the isolation of data/instructions in the non-secure segments of the super-TEE during the offline profiling step. More importantly, the software developer must utilize an existing automated approach [85] to maintain the separation of resource between two CAs. The fused CA and its corresponding fused TAs form a super-TEE.

Figure 6.4 illustrate an example code transformation [85] to construct a super-TEE. Lines $1-19$ show two independent tasks τ_1 and τ_2, both of which require TEE execution. In task τ_1, funcA() and funcC() form the non-trusted execution sections, while funcB() requires TEE execution. Similarly, for τ_2, funcX() and funcZ() constitute the non-secure execution sections, while funcY() requires TEE execution. Lines $21-36$ show the re-factored code (CA) where tasks τ_1 and τ_2 are fused to form a super-TEE while maintaining its logical correctness. Finally, Lines $38-45$ show the corresponding trusted segments of tasks τ_1 and τ_2 which have been partitioned into single TAs for TEE execution. The Super-TEE calls TAs using TEE client APIs as demonstrated in lines $29-30$ (Figure 6.4).

```
1 main(){ //Task τ1
2 funcA(); //Non−trusted code
3 funcB(); //Call funcB()
4 ...
5 funcC();//Rest of non−trusted code
6 }
7 //Annotated Segment: assume requires TEE execution
8 int funcB(){
9 ... } //Segment END
10
11 main(){ //Task τ2
12 funcX(); //Non−trusted code
13 funcY(); //Call funcY()
14 ...
15 funcZ();//Rest of non−trusted code
16 }
17 //Annotated Segment: assume requires TEE execution
18 int funcY(){
19 ... } //Segment END
20
21 //Super-TEE
22 main(){
23 funcA(); //Non−trusted code: Task τ1
24 funcX(); //Non−trusted code: Task τ2
25 //Start TEE execution using TEE client APIs
26 TEEC_InitializeContext(...);//Set up TEE context
27 TEEC_OpenSession(...); //TEE entry
28 //Call trusted applications (TAs)
29 TEEC_InvokeCommand(funcB,...);//TEE call: Task τ1
30 TEEC_InvokeCommand(funcY,...);//TEE call: Task τ2
31 TEEC_CloseSession(&sess);
32 TEEC_FinalizeContext(&ctx); //End TEE execution
33 ...
34 funcC()//Rest of non−trusted code: Task τ1
35 funcZ()//Rest of non−trusted code: Task τ2
36 }
37
38 //TA code: run in trusted execution environment
39 static TEE_Result funcB(...){
40 ... //funcB() body
41 return TEE_SUCCESS;}
42 ...
43 static TEE_Result funcY(...){
44 ... //funcY() body
45 return TEE_SUCCESS;}
```

Figure 6.4: Code snippet of two example tasks τ_1 and τ_2 which need to be re-factored for TEE execution, and fused into super-TEE and its corresponding trusted segments (TAs).

6.3.2 Security Impacts

We make the observation that TEE overhead can be reduced with minimal (possibly no) security impact on trusted execution since our approach does not require modifications to TEE. Specifically, each trusted segment executes within a secure isolated context (TEE), which is implemented and maintained within the trusted OS. All instructions and meta-data specific to its execution is contained within its own secure thread in the trusted OS. Since each TEE execution is initiated from the non-trusted side, it requires a switch back from the trusted environment. In such a case, the trusted OS (i) maintains the context of the secure execution (not visible to the non-trusted OS) before (ii) resetting the execution environment while transferring the control over to the non-secure side. Constructing a super-TEE by fusing two separate trusted execution segments, thus, simply implies that both will run inside the same trusted environment, however, on isolated secure threads. While this ensures that an attacker in control of a task that includes TEE execution cannot subvert either the original trusted execution sections or the fused ones it, nonetheless, potentially allows for side channel attacks [66, 81] in some task instances (e.g., for fused tasks that accept attacker-controlled input) because the context (including registers and caches) is not securely removed between two consecutive TEE sections of the super-TEE. Next, we will formally define the threat model for the specific security vulnerability induced on super-TEEs from the non-secure side.

Threat Category

We consider cache based side-channel attacks, which is a common vulnerability in TEEs. An inherent architectural feature of caches is the considerable difference in the data access times for cache hits and misses. Cache based side-channel attacks exploit this timing anomaly to

mask their approach, thereby making them hard to detect and mitigate.

In general, cache side-channel attacks can be time-driven, trace-driven or access-driven. Each category has the same objective, but differ in the granularity of their approach. In trace-driven approach [97], the attacker not only has the capability to observe and record each cache access and its outcome during a victim's execution context, but can also detect the information being accessed and/or modified. In time-driven approach [97], the attacker uses a comparatively coarse granularity of approximating the overall number of cache hits and misses during the execution context of a victim. Conversely, access-driven approach [96] leverages spatial locality in cache access patterns to detect the location where the vulnerable information resides, and is being processed from, during the victim's execution context.

In an ideal case, an attacker has the capability to (1) observe each cache access during the victim's execution context, (2) manage the data footprint being processed and/or modified, and, (3) deduce the memory location of each access, and thereby the region of interest for critical information. However, in real-world scenarios, attackers do not have such room for flexibility. Therefore, an indirect access-based attack technique [131] measures the system's cache access overhead by timing its own cache hits and misses, while competing with the victim to account for resource contention (e.g., memory bandwidth). Another timing-based attack technique [23] records the overall duration of execution for the victim's region of interest, and leverages the information to devise an threat.

Threat Identification and Mitigation Strategy

Fusing two trusted execution sections together to run within the same TEE execution context is specifically susceptible to access-based cache side-channel attacks, where an attacker accesses its own memory region, and captures the associated access times to infer the victim's

use of shared cache. Let us consider that two tasks τ_A and τ_B, the victim and the attacker respectively, require TEE execution, and are fused to form a super-TEE task. Both the tasks share the same secure cache space. Let us assume that an attacker can control the input to the non-TEE (CA) executable of task τ_B, which runs directly after task τ_A, which itself has sensitive data stored in the cache. Furthermore, assume that τ_B performs computations on the input data that results in cache access and that the attacker understands how this occurs (i.e., attacker has access to tasks' τ_A and τ_B's code). Therefore, if the compromised non-TEE client application (CA) executable that the attacker controls, from the non-secure environment, allows it to make timing measurements, it may be possible for the attacker to recover data from the cache [66, 81] based on how long computations take for task τ_B.

A possible solution to mitigate this threat would require flushing of cache (and clearing of CPU registers) between execution of two separate TEE sections for tasks τ_A and τ_B. Experimental results on our hardware test bed (Rpi 3B) shows an overhead of $22\,\mu s$ to flush out up to 4 KB of secure cache data.

6.3.3 Super-TEE Task Model

While discussing the construction of a super-TEE in the previous section, we focused on the logical correctness of the tasks. We now examine the timing aspects of a super-TEE in this section. Let us revisit our motivating example in Section 6.2 where the corresponding synchronous task set is shown in Table 6.1. Let us consider fusing τ_A and τ_B together to form a super-TEE. Since τ_A and τ_B have different periods, there are two challenges associated with upholding temporal correctness: (1) executing the super-TEE with τ_B's period (17 time units) may be logically incorrect since τ_A (with smaller period) needs to run more frequently, and, (2) executing the super-TEE with τ_A's period (16 time units) may result in unnecessary

resource usage since τ_B (with larger period) needs to run less frequently. Therefore, the challenge is to ascertain the time intervals where both τ_A and τ_B execute so that we can run the super-TEE, while at other times we run only τ_A since it is the task with a lower period. We model a super-TEE as a variant of the multi-frame task model [75]. A super-TEE task τ_{ij} constructed by fusing synchronous tasks τ_i and τ_j is defined as (T_{ij}, C_{ij}^{peak}, C_{ij}^{nml}, l_{ij}). T_{ij}, defined as the period of the super-TEE task τ_{ij}, is set as $\min\{\mathrm{T_i}, \mathrm{T_j}\}$. In our example task set (Table 6.1), $T_{AB} = 16$, since the period of task τ_A is shorter than task τ_B. The execution time parameter C_{ij}^{peak} corresponds to the WCET of a frame when the fused TEE section is running, and is calculated using the equation 6.2. Similarly, C_{ij}^{nml} corresponds to the WCET of a frame when the fused TEE section is not running, and is calculated using the equation 6.3. From Equations 6.2–6.3, we know that for each job instance of the task-tuple $\{\tau_A, \tau_B\}$ which coincides with the fused TEE section runs with a WCET of $C_{\text{AB}}^{\text{peak}} = 15$ time units, as discussed in Section 6.2, while the job instance which corresponds to non-fused TEE execution has a WCET of $C_{\text{AB}}^{\text{nml}} = 8$ time units , i.e., the WCET of τ_A, the task with the smaller period.

$$C_{ij}^{\text{peak}} = \sum_{q=1}^{2} \left[C_i^q + C_j^q \right] + \upsilon_i + \upsilon_j - (\upsilon_j^s + \upsilon_j^d). \tag{6.2}$$

$$C_{ij}^{\text{nml}} = \sum_{q=1}^{2} C_i^q + \upsilon_i. \tag{6.3}$$

$$l_{ij} = \left\lfloor \frac{T_j}{T_{ij}} \right\rfloor. \tag{6.4}$$

The parameter l_{ij} captures the minimum inter-peak frame distance such that every l_{ij} consecutive frames contain at most one peak frame. In our example, since at every 16 time units interval, there must be at most one C^{peak} frame to account for each job instance of tasks τ_A

Table 6.2: Task Set 2

Task (τ)	C	T
τ_A	1	3
τ_B	1	7

Table 6.3: Modified Task Set 2

Task (τ)	C^{peak}	C^{nml}	T	l
τ_{AB}	1.7	1	3	2

and τ_B, we set $l_{\text{AB}} = 1$ (Equation 6.4). Tasks which are not super-TEEs have $C^{\text{peak}} = C^{\text{nml}}$. Now, for each task $\tau_i = (T_i, C_i^{\text{peak}}, C_i^{\text{nml}}, l_i)$ in the task set $\Psi = \{\tau_i : i = 1, \ldots, n\}$ the total task set utilization is given by

$$U_T = \sum_{i=1}^{n} \left[\frac{C_i^{nml}}{T_i} + \frac{C_i^{\text{peak}} - C_i^{\text{nml}}}{l_i T_i} \right] \tag{6.5}$$

The revised real-time parameters of the tasks in the task set (Table 6.1) is represented by the tuple $(16, 15, 8, 1)$, where $T_{AB} = 16$, $C_{AB}^{peak} = 15$, $C_{AB}^{nml} = 8$, and $l_{AB} = 1$.

Example #1:

Let us consider two tasks with co-prime time periods. The real-time parameters of the tasks in this task set is given in Table 6.2. We will now help the readers walk-through the process of generating real-time parameters for this task τ_{AB}. We consider that each task has trusted segments that need to run in TEE. The upper bound on the worst-case execution time $(\sum_{q=1}^{2} C_i^q + \upsilon_i)$ for both tasks τ_A and τ_B is set at 1. Also, let us assume that the total SMC overhead ($\upsilon_{\text{i}}^{\text{s}}$ and $\upsilon_{\text{i}}^{\text{d}}$) for both tasks is 0.3 time units. Therefore, using Equations 6.2–6.3, we have $C_{AB}^{\text{peak}} = 1.7$ and $C_{AB}^{\text{nml}} = 1$. Similarly, T_{AB} is set to 3 ($\min\{\text{T}_\text{A}, \text{T}_\text{B}\}$). From Equation 6.4, we know that l_{AB} equals to 2, and the super-TEE maintains temporal correctness, as long as each $(T_{AB} \cdot l_{\text{AB}})$ time interval contains at most one C^{peak} frame (Figure 6.5). Note that at time 0, a job of τ_A and τ_B has each been released and so the super-TEE can execute its C_{AB}^{peak} frame (fused TEE section). In contrast, at time 6, the super-TEE will have to run

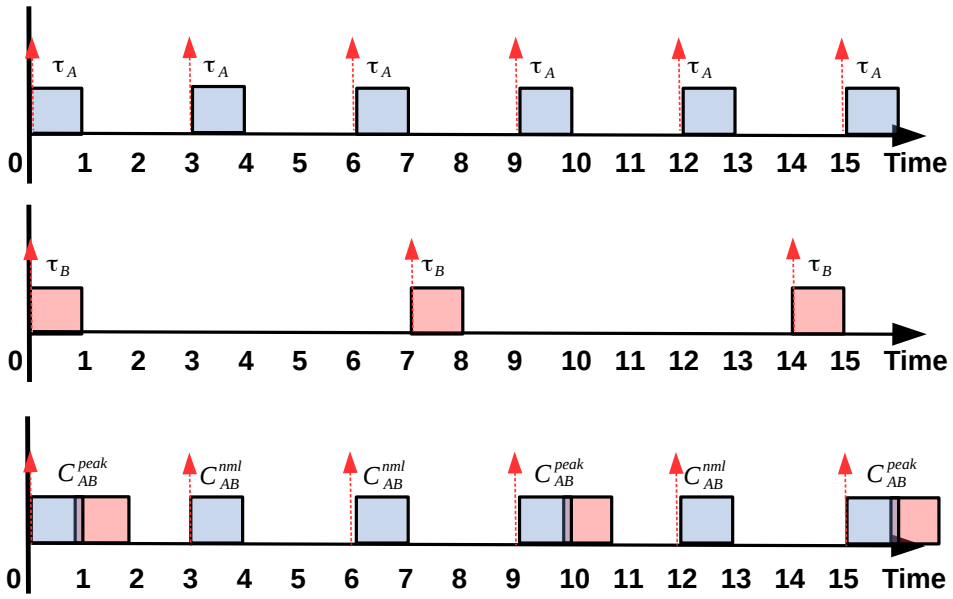


Figure 6.5: A example execution instant of the super-TEE (Table 6.3).

Table 6.4: Task Set 3

Task (τ)	C	T
τ_A	1	3
τ_B	1	4

Table 6.5: Modified Task Set 3

Task (τ)	C^{peak}	C^{nml}	T	l
τ_{AB}	1.7	1	3	1

its C_{AB}^{nml} frame since a new job instance of τ_B is yet to be released. Hence, the next C_{AB}^{peak} frame can only be formed at time 9. The same pattern is repeated after every hyperperiod of τ_{AB} (the least common multiple of the periods of τ_A, τ_B). The modified task set is shown in Table 6.3.

6.3.4 Feasibility of Super-TEE Candidates

The main objective for fusing TEE execution sections together is to save CPU cycles every time an instance of a super-TEE is scheduled as shown in Figure 6.3. While fusing the TEE execution sections of tasks reduce the collective WCET, such a modification may change

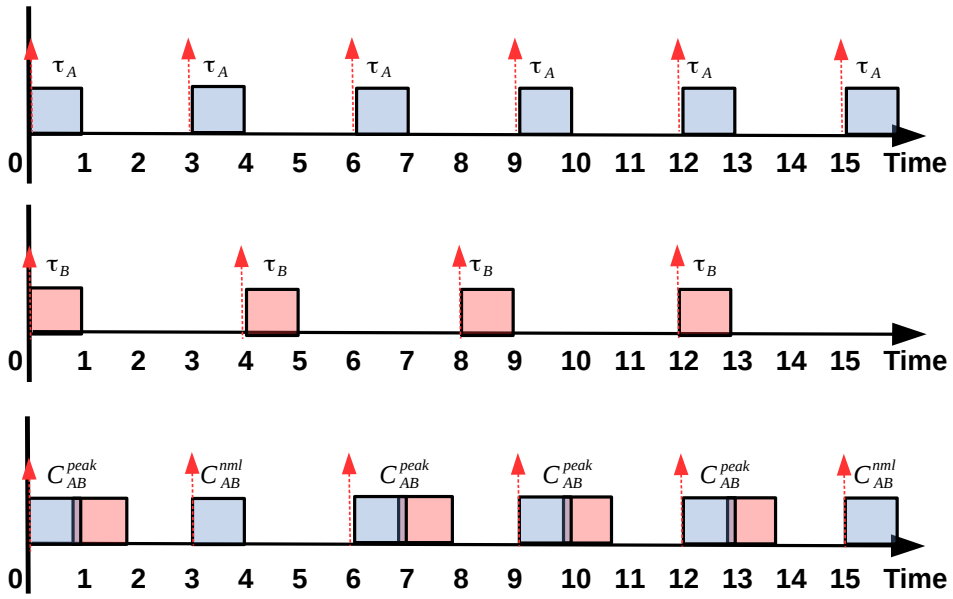


Figure 6.6: A example execution instant of the super-TEE (Table 6.5).

the collective period of the tasks under consideration, thereby, increasing the total system utilization. We present a schedulability analysis to determine whether a super-TEE candidate is in fact feasible, i.e, the super-TEE candidate will not cause an increase in the total system utilization. A task-tuple, i.e., super-TEE candidate, that fails to meet the condition stated in Theorem 6.1 is removed from further consideration. That is, the tasks will execute independently without having their TEE sections fused as shown in Algorithm 6.2.

Observe that in Example #1 (Section 6.3.3), even though a new job of task τ_B is released at time 7 (Figure 6.5), it can only start within the next frame of super-TEE (C^{peak}) at time 9. We now discuss a requirement with regards to the delay between the release time and the start time for a task that any super-TEE must follow. Let us consider the task set in Table 6.4 where an execution instant of the modified task set (Table 6.5) after super-TEE construction is illustrated in Figure 6.6. At time 6, if a higher priority job preempts the super-TEE job for longer than 0.3 time units, task τ_B within C^{peak} frame will miss its deadline even though the super-TEE itself will not. To guarantee the timeliness of individual

tasks τ_A and τ_B that forms a super-TEE, we must have at least one job of τ_A completely contained within each period of τ_B. We start with the following lemma.

Lemma 6.1. Let us consider a super-TEE candidate, which consists of the task-tuple $\{\tau_i, \tau_j\}$. If $T_i \leq T_j$ and $2 \cdot T_i - gcd(T_i, T_j) \leq T_j$ where gcd() denotes the greatest common divisor, then each job of the super-TEE candidate with period T_i is guaranteed to meet the individual deadlines of tasks τ_i and τ_j.

Proof. We leverage cyclic scheduling of synchronous periodic tasks [13], which are invoked in frames. Each frame of length m that invokes a job of a task τ_k must start after the job arrival and end before the job's period to guarantee schedulability, also given by,

$$m + (m - gcd(m, T_k)) \leq T_k, \tag{6.6}$$

where T_k is the period of task τ_k and, gcd is the greatest common divisor. For our super-TEE candidate $\{\tau_i, \tau_j\}$, the period is set at $\min(\mathrm{T_i}, \mathrm{T_j})$ (Section 6.3.3). To guarantee the timeliness of individual tasks τ_i and τ_j that form a super-TEE, we must have at least one task of τ_i (i.e., equivalent to a frame) completely contained within each period of execution of τ_j. Therefore, Equation 6.6 can be rewritten as $2 \cdot T_i - gcd(T_i, T_j) \leq T_j$, and the lemma holds. ■

Lemma 6.2. Let us consider a super-TEE candidate, which consists of the task-tuple $\{\tau_i, \tau_j\}$. Then, $C_{ij} < C_i + C_j$.

Proof. Let us assume that the upper bound on the worst-case computation time by fusing the task-tuple $\{\tau_i,\tau_j\}$ into a super-TEE τ_{ij} is denoted by C_{ij}. From Equation 6.2, we have,

$$C_{ij} = C_{ij}^{peak} = \sum_{q=1}^{2} C_i^q + \upsilon_i + \sum_{q=1}^{2} C_j^q + \upsilon_j - (\upsilon_j^s + \upsilon_j^d),$$

$C_i = \sum_{q=1}^{2} C_i^q + \upsilon_i$, and $C_j = \sum_{q=1}^{2} C_j^q + \upsilon_j$. We define $C_j' = \sum_{q=1}^{2} C_j^q + \upsilon_j - (\upsilon_j^s + \upsilon_j^d)$. Since $C_j' < C_j$, we have $C_{ij} < C_i + C_j$, and the lemma holds. ■

We are now ready to present a sufficient condition for schedulability of a super-TEE candidate for a given task set.

Theorem 6.1. Let us consider a task set Ψ that is deemed schedulable according to the LL limit (Equation 6.1). Further, let us consider a super-TEE candidate, which consists of the task-tuple $\{\tau_i, \tau_j\}$, where $\tau_i, \tau_j \in \Psi$. Let us refer to this task tuple as τ_{ij}, which can be modeled as discussed in Section 6.3.3. If lemma 6.1 holds, and $C_i \geq C_j$, $T_i \leq T_j$, and $\frac{C_j}{C_j'} \geq \frac{T_j}{T_i}$, where $C_j' = C_j - (\upsilon_j^s + \upsilon_j^d)$, then $\Psi' = (\Psi \cup \tau_{\text{ij}}) - \tau_i - \tau_j$ is also schedulable per Equation 6.1.

Proof. According to Lemma 6.2, we have $C_{ij} < C_i + C_j$. Given that Ψ is RM schedulable according to Equation 6.1, Ψ' is also schedulable if $U(\tau_{\text{ij}}) \leq U(\tau_i) + U(\tau_j)$. We know that

$$U(\tau_{\text{ij}}) = \frac{C_i + C_j'}{T_i}$$

Given that $\frac{C_j}{C_j'} \geq \frac{T_j}{T_i}$, then we can replace C_j' in the above equation with $\frac{C_j \cdot T_i}{T_j}$, and get,

$$\begin{aligned} U(\tau_{\text{ij}}) &\leq \frac{C_i + \frac{C_j \cdot T_i}{T_j}}{T_i} \\ &\leq \frac{C_i \cdot T_j + C_j \cdot T_i}{T_j \cdot T_i} \\ &\leq \frac{C_i}{T_i} + \frac{C_j}{T_j} \\ &\leq U(\tau_{\text{i}}) + U(\tau_{\text{j}}). \end{aligned}$$

■

Table 6.6: Original Task Set

Task (τ)	C^1	υ	C^2	C	T
τ_1	0.2	0.7	0.1	1	2
τ_2	0.6	0	0	0.6	8
τ_3	0.1	0.8	0.1	1	2

An important consequence of Theorem 6.1 is that if the original task set is schedulable, and we only construct a super-TEE when Theorem 6.1 holds, the modified task set is also guaranteed to be schedulable. While we cannot at this point provide any guarantee on the schedulability of the modified task set if the original task set is not schedulable, our approach can sometimes make the task set schedulable as shown in the example below. We will also show that our approach can in fact improve schedulability under many scenarios as discussed in Section 6.6.

Example:

Let us consider the task set in Table 6.6, which fails the RM schedulability test according to Equation 6.1. Applying our approach, the modified task set is shown in Table 6.8 and contains a task tuple $\{\tau_1, \tau_3\}$ fused to form a super-TEE according to Section 6.3.5, and an individual task τ_2. Although the super-TEE has a modified worst-case execution time and period, it passes the feasibility test outlined above. Since the utilization of this modified task set falls within the LL limit (Equation 6.1), the tasks are guaranteed to be schedulable using RM.

Table 6.7: Candidate super-TEE profiles

Task-tuple	$ Footprint	# Concurrent jobs	Execution Overlap	Rank
$\{\tau_1, \tau_2\}$	20%	1	0.6	2
$\{\tau_1, \tau_3\}$	10%	4	1	1
$\{\tau_2, \tau_3\}$	20%	1	0.6	2

Table 6.8: Optimized Task Set

Task (τ)	C^{peak}	C^{nml}	T	l
$\{\tau_1, \tau_3\}$	1.2	1	2	1
τ_2	0.6	0.6	8	–

6.3.5 Finding Super-TEE Candidates

Now that we have explained how to construct a super-TEE and find its aggregated utilization, we turn our attention to selecting the actual tasks best suited to form a super-TEE. The first step is to ensure that constructing a super-TEE does not negatively impact the schedulability of the entire system, as discussed in the previous section. Therefore, we apply Theorem 6.1 to all possible combination of task-tuples $\{\tau_i,\tau_j\}$ to check for a feasible taskset. Once the feasibility criterion is met, every possible combination of task-tuple $\{\tau_i,\tau_j\}$, $\tau_i, \tau_j \in \Psi$, is profiled for its execution pattern, concurrent job instances, maximum execution overlap duration, and combined secure cache footprint. This is a computationally intensive process, but only needs to be performed once offline. Given all the combination of task-tuples, our goal is to eliminate infeasible task-tuples and rank the remaining ones. Task-tuples whose secure cache footprint exceeds the maximum cache size of a core is removed from further consideration; architecturally, the execution overhead of an application significantly reduces if it experiences a low cache miss rate. By only considering task-tuples that are within the cache limit, we avoid unnecessary cache misses, thereby reducing memory fetches during

application runtime and improve temporal predictability.

Since there may be a large number of task-tuples that are within the cache limit, we propose to prioritize the task-tuples by their concurrent job instances. For example, from Figure 6.1(b), we can graphically deduce that a task-tuple $\{\tau_1, \tau_3\}$ has a total of 4 concurrent job instances within the hyperperiod, while task-tuple $\{\tau_1, \tau_2\}$ has 1. Since the system saves CPU cycles every time the super-TEE of a task-tuple executes (Lemma 6.2), the higher the frequency of overlapping execution sections, the larger the increase in CPU cycle savings. For instance, in Table 6.7, since $\{\tau_1, \tau_3\}$ has $\max\{\#\ Concurrent\ jobs\}$, it receives rank 1 (highest priority). Ties are broken in favor of task-tuples with the smallest interval of overlapping execution sections, as smaller overlaps between execution intervals of two tasks indicate higher compatibility for sequential execution of the tasks under consideration [112]. Table 6.7 shows the ranked list of all task-tuples from our example task set (Table 6.6).

The steps to determine the tasks to fuse together are shown in Algorithm 6.1. The $TupleList$ is the set of all task-tuples. Task-tuples whose cache footprint exceeds $FootprintLimit$ are discarded. For each of the remaining task-tuples, SORT_CONCURRENT_COUNT() calculates the number of concurrent jobs if the tasks in the task-tuple were run in isolation till their hyperperiod, and sorts the tuples in a non-increasing order. Finally, ties are broken by comparing the size of overlaps to create a ranked $TupleList$.

6.4 CT-RM Scheduling Algorithm

To the best of our knowledge, there are no existing approaches to schedule super-TEEs in a hard real-time system while providing deadline guarantees. As such, we leverage an existing fixed-priority rate-monotonic scheduling policy (RM-FF) to schedule super-TEEs, along with other real-time tasks, on multicore systems. We opt for partitioned scheduling

Algorithm 6.1 Task Fusion

```
function Tuple_Optimization(tupleList)                          ▷ Rank task tuples
    for {τ_i, τ_j} ∈ tupleList do
        if Cache_Footprint({τ_i, τ_j}) > FootprintLimit then
            Remove_Tuple(tupleList, τ_i, τ_j)
    Sort_Concurrent_Count(tupleList)                            ▷ Non-increasing
    for tuple1, tuple2 ∈ tupleList do
        if tuple1.overlap == tuple2.overlap then
            Sort_Overlap_Size(tuple1, tuple2)                   ▷ Non-decreasing
function Find_Candidate(τ_i, τ_j)                               ▷ Check feasibility: Theorem 1
    if C_j/C'_j ≥ T_j/T_i and 2 · T_i − gcd(T_i, T_j) ≤ T_j then return 1
    else return 0
▷ Generate multi-frame task parameters for feasible super-TEEs
function Construct_SuperTEE(tupleList)
    for {τ_i, τ_j} ∈ tupleList do
        if Find_Candidate({τ_i, τ_j}) then
            Assign C_ij^peak using Equation 6.2
            Assign C_ij^nml using Equation 6.3
            Assign l_ij using Equation 6.4
        else
            Remove_Tuple(tupleList, τ_i, τ_j)
function main(tupleList)
    Construct_SuperTEE(tupleList)
    Tuple_Optimization(tupleList)
    return
```

since it has the advantage of reducing the multiprocessor scheduling problem to scheduling on individual processors. In addition, since our task set consists of normal real-time tasks and super-TEEs, we modify RM-FF by changing the task partitioning policy. The advantages of our modifications are apparent when discussed in Section 6.5.

We first define a $TupleList$, which consists of a list of task-tuples, e.g., (τ_i, τ_j), where τ_i and τ_j form a super-TEE (Section 6.3.5). We sort task-tuples in a non-decreasing order of periods to obtain the worst-case critical instant [44]. We divide the task-to-core assignment step into two phases (Algorithm 6.2). In the first phase, we leverage the existing RM First-Fit (RM-FF) partitioning policy, where tasks are assigned to a core until it is no longer RM schedulable, after which, the next core is considered [44], to assign tuples (super-TEEs) to cores. We set per-core admissible utilization bound to the LL limit (Equation 6.1). (A worst-case response time analysis can be used instead and is left for future work.) Lines [4-8] in Algorithm 6.2 performs the first phase of task-to-core mapping using the first-fit policy. In the second phase, we turn our attention to $RemTaskList$, which consists of tasks which are not part of $TupleList$. Lines [11-14] in Algorithm 6.2 performs the second phase of task-to-core mapping, again, using the first-fit policy after having sorted the tasks in a non-decreasing order of periods.

Algorithm 6.2 Task-to-Core Mapping

```
1: function Task_To_Core(TupleList, TaskList, Cores)
2:     U_max ← LL limit                          ▷ Check for LL bound: Equation 6.1
3:     for {τ_i, τ_j} ∈ TupleList do             ▷ Assign tuple to core
4:         {τ_i, τ_j}.Core ← First_Fit({τ_i, τ_j}, U_max, Cores)
5:         if ¬{τ_i, τ_j}.Core then              ▷ If tuple assignment fails
6:             Remove_Tuple(TupleList, τ_i, τ_j)
7:     RemTaskList ← TaskList − TupleList
8:     RemTaskList ← Sort_Increasing_Period(RemTaskList)
9:     ▷ Assign remaining tasks to cores
10:    for task ∈ RemTaskList do
11:        task.Core ← First_Fit(task, U_max, Cores)
12:        if ¬task.Core then                    ▷ If task assignment fails
13:            exit()
```

Table 6.9: Summary of experimental results of synthetic benchmark (Figure 6.4) running on Rpi 3B using ct-RM and RM-FF with respect to percentage of feasible task sets as a function of task set utilization.

Util (%)	RM-FF Feasible task set (%)	ct-RM Feasible task set (%)
100	100	100
150	100	100
200	87	96
250	72	86
300	21	48

6.5 Experiments

In this section, we evaluate the benefits of our proposed approach by scheduling real-time synthetic benchmarks on an actual hardware platform.

6.5.1 Experimental Setup

For our hardware testbed, we use the Raspberry Pi 3B (Rpi 3B), a small computer with quad-core ARM Cortex A53 processor, 1 GB LPDDR RAM, and numerous sensors. It can run Linux and other non-trusted operating systems. It also extends support for ARM TrustZone. We used the Linux RT_PREEMPT Kernel v4.6.3 to build our prototype ct-RM by modifying the Linux real-time scheduling class SCHED_DEADLINE. We run the modified real-time Linux OS (as non-trusted OS) along with OP-TEE OS [2] (as TEE) on the Rpi 3B that extends the support for ARM v8 embedded virtualization. We also designed a task set generator to create synthetic benchmark applications as shown in Figure 6.7. The inputs to the said task set generator are util_factor, tee_tasks and tee_exec, where util_factor represents the task set utilization level, tee_tasks denotes the number of tasks with TEE requirements in each task set and, tee_exec is the maximum percentage of worst-

case execution time of each task that has to be assigned for trusted execution in a TEE environment.

For the experimental results presented in Table 6.9, the task sets were generated over a range of utilization levels (util_factor = 100%, 150%, 200%, ..., 300%) and, for each utilization level, we generated 100 task sets, each of which has a random number of tasks. The period (T_i) and worst-case execution time (C_i) of each task are randomly generated so long as the overall utilization of the task set remains within the corresponding utilization level. The worst-case execution time of a task τ_i is upper bounded by $\sum_{q=1}^{2} C_i^q + \upsilon_i$, where C_i^q denotes each non-secure execution segments, and υ_i denotes the trusted execution interval. Each task set consists of (tee_tasks =) 60% of the tasks with TEE requirements and 40% ordinary real-time tasks. The trusted execution duration (υ_i = tee_exec) of each TEE task is set at 60% of the worst-case execution time. The secure cache size limit is set to a high 90% of the maximum cache size to remove the effect of hardware-specific cache limit. We use all the 4 cores of Rpi 3B to run our experiments. Each experiment is carried out for one hyperperiod, the least common multiple of the periods of all the tasks in a task set. The results in Table 6.9 show an improved usable utilization bound by 12%, on average by ct-RM over RM-FF across all utilization levels (100%, 150%,..., 300%).

Case Study:

We explain the implementation details for scheduling ct-RM on tasks whose attributes are shown in Table 6.8, along with another real-time task τ_4 with attributes $(2, 1, 1, -)$ (see Section 6.3.3). The task set is scheduled on a 4-core platform. All the tasks realize context-switching between non-trusted and trusted environments through an INTERRUPTABLE sleep duration. Each task starts as a completely fair scheduler (CFS) task which eventually switches to an RM task, and is distinguished from the other by the scheduler through a bit

combination realized through a tuple (cpu, OPTEE_on, OPTEE_task) and communicated through a custom syscall to the kernel. The cpu denotes the assigned core, OPTEE_on categorizes ct-RM as the scheduling policy for the task and the OPTEE_task bit distinguishes a Super-TEE task ($\{\tau_1, \tau_3\}$) from other real-time tasks (with or without TEE requirements). Note, that ct-RM prohibits any task migration. Hence, we set the scheduler flag NR_CPUS_ALLOWED to #1 in the Linux scheduler. This informs the scheduler that the current task will never be up for migration. We validate the task-to-core assignment, the desired real-time characteristics and application's flow of execution by tallying the kernel log time stamps. Experiment log reveals that all the tasks complete their execution by the deadlines and behave as expected.

6.6 Simulations

Since architecture-specific design of Rpi3B constrains the maximum available secure cache dedicated for TEE usage, our hardware testbed is limited by the number of TEEs that can run simultaneously. Therefore, we extend the validation of our proposed approach (ct-RM) and assess its real-time performance against RM-FF in a simulated environment on randomly generated task sets. Similar to our hardware experiments, we use the task set generator over a range of utilization levels (util_factor = 100%, 150%, 200%, 250%,..., 1000%). For each utilization level, we generated 100 task sets, each of which has a random number of tasks, of which tasks with TEE sections have tee_exec ranging between $30\% - 90\%$. For the reported simulation result, each task set consists of (tee_tasks =) 60% tasks with TEE requirements. The secure cache size limit is set to a high 90% of the maximum cache size to remove the effect of hardware-specific cache limit. Each simulation run tests the feasibility of the generated task sets using the steps listed in Algorithm 6.2.

```
1 #define SCHED_DEADLINE 6 //Modified SCHED_DEADLINE for CT−RM
2 void *inc_x(void *x_void_ptr){ //Normal RM task body
3 ...
4 attrX.sched_policy= SCHED_DEADLINE;
5 ret = sched_setattr(0, &attrX, flags); //set CT−RM params
6 ...
7 while(++(*x_ptr) < 100); //normal computation
8 ...
9 return NULL;}
10 void *inc_y(void *y_void_ptr){//super−TEE task body
11 funcA(); //Non−trusted code: Task τ1
12 funcX(); //Non−trusted code: Task τ2
13 usleep (...) ; //simulate self−suspension for data dependency
14 ...
15 funcC()//Rest of non−trusted code: Task τ1
16 funcZ()//Rest of non−trusted code: Task τ2
17 }
18 void *inc_z(void *z_void_ptr){//fused TEE section
19 *'\colorbox{light-gray}{//Start TEE execution}'*
20 TEEC_InitializeContext(NULL, &ctx);
21 TEEC_OpenSession(&ctx,&sess,&uuid,..);
22 TEEC_InvokeCommand(funcB,...);//TEE call: Task τ1
23 TEEC_InvokeCommand(funcY,...);//TEE call: Task τ2
24 TEEC_CloseSession(&sess);
25 TEEC_FinalizeContext(&ctx);
26 *'{\colorbox{light-gray}{//End TEE execution}'*
27 ...}
28 int main(){ //Main CFS task to spawn other SCHED_DEADLINE tasks
29 syscall(288, 1, 1, 0); //independent RT task to run on core #1
30 pthread_create(&inc_x_thread, NULL, inc_x, &x);
31 syscall(288, 2, 1, 0); //super−TEE task to run on core #2
32 pthread_create(&inc_y_thread, NULL, inc_y, &x);
33 while (...) {}//wait for TEE execution
34 syscall(288, 2, 1, 1); //fused TEE section to run on core #2
35 pthread_create(&inc_z_thread, NULL, inc_z, &z);
36 //wait for tasks to end
37 return 0; }
```

Figure 6.7: Code snippet of our representative benchmark application modeling the super-TEE shown in Figure 6.4.

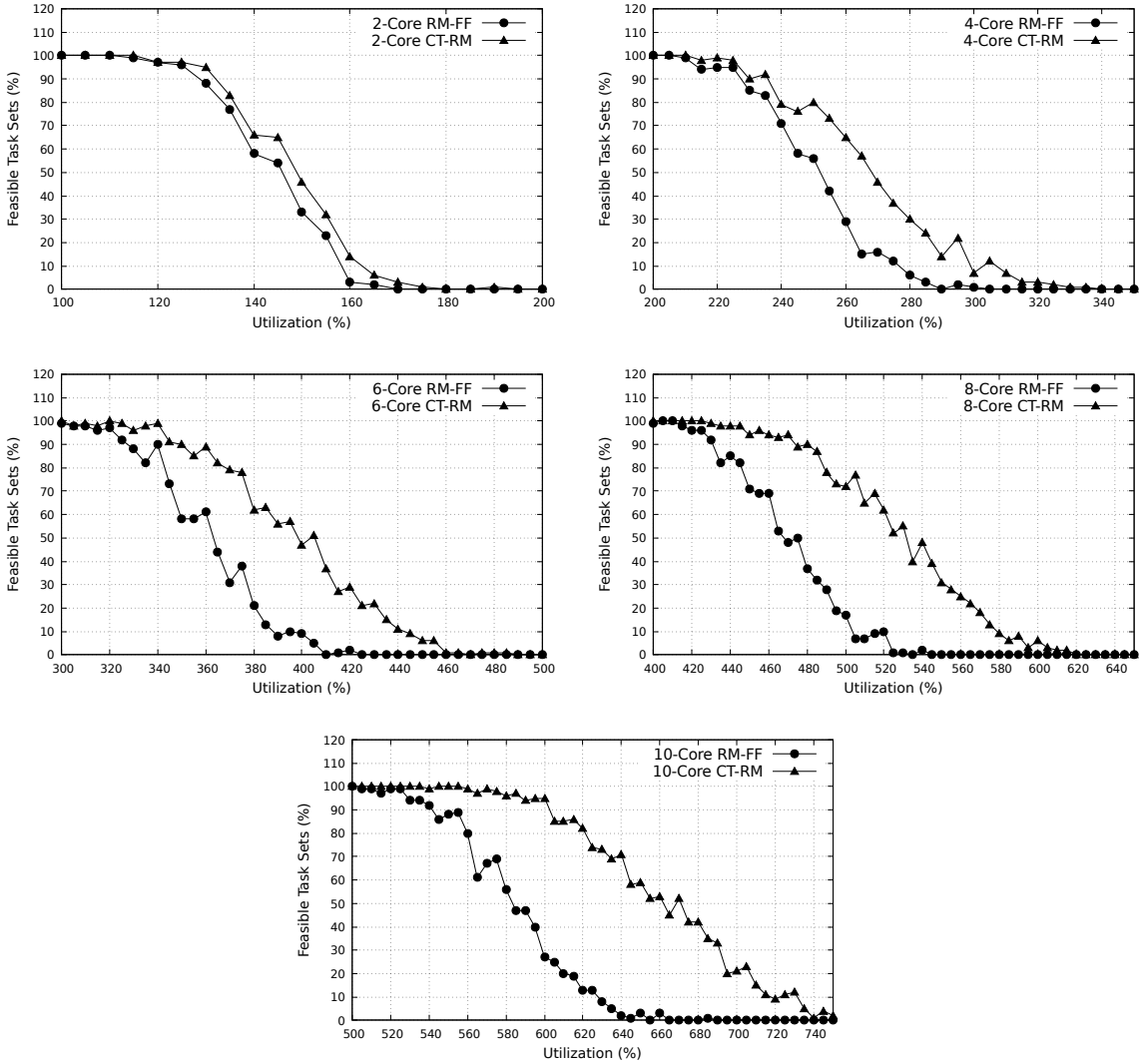


Figure 6.8: Simulation results showing the number of feasible synchronous task sets as a function of system utilization demand with scaling core count.

Figure 6.8 reports the average results comparing the performance of ct-RM against RM-FF in terms of the number of feasible synchronous task sets as a function of utilization levels with scaling cores. Our results indicate that the region of improvement with our approach over partitioned RM-FF with scaling utilization levels over increasing core count widens, indicating a scalable solution. While we observe a comparable performance between RM-FF and CT-RM for a 2-core system (an average of 4% improvement in task set feasibility across all utilization levels), the effectiveness of our solution is more pronounced in an 8-core and 10-core system (an average of 23% and 34% improvement in task set feasibility respectively across all utilization levels). Overall, the trend shows an improved feasibility of up to 38% and, 18% on average over partitioned RM-FF across all utilization levels.

Since this work targets hard real-time systems, we further test the applicability of our solution on tasks with harmonic periods, a subset of synchronous task sets. Harmonic task sets are widely used in industry applications, ranging from cyber-physical systems [24, 76, 121] to control systems [51], as they allow a 100% utilization to be realized when using a fixed-priority scheduling policy [58]. Comparing the performance of ct-RM in terms of the number of feasible harmonic task sets as a function of utilization levels with scaling cores show improved feasibility of 20% on average over partitioned RM-FF across all utilization levels.

For our simulation results, we identify the worst-case scenario for our proposed solution when a task set (1) has minimum or no tasks with TEE requirements, and, (2) tasks with TEE requirements have shortest WCET of trusted execution sections. By extension, this complies with a task set which is similar to a normal real-time task set, consisting of tasks without TEE execution requirement. Such a task set may be scheduled feasibly with the existing RM-FF scheduling algorithm. Therefore, the worst-case results for our solution is equivalent to the results obtained by scheduling task sets that adhere to the worst-case scenario with RM-FF. The probability of obtaining a feasible schedule for such a task set

using our proposed framework is exactly the same as the improvement we have reported over RM-FF.

6.7 Related Work

Security for embedded hardware and/or software is a main focus of recent work [36, 52]. For real-time systems, the emphasis is usually on the trade-off between real-time constraints and security levels [67, 86]. Hasan et al. [59] created a security policy to realize a fixed-priority sporadic server. Our work, however, is orthogonal to existing work and can be used in conjunction rather than instead of.

ARM TrustZone is a widely used platform-level security solution for many real-time embedded systems with security requirements. For instance, virtualization solutions [50, 106] allow Linux OS and TEE to run simultaneously while maintaining real-time performance. Pinto et al. [104] used the ARM TrustZone to run a low priority thread of a real-time OS over secure virtualization. OP-TEE [2], which is an open source port of TEE-specific design for Linux running on the ARM hardware platform, was used to run trusted sections alongside a real-time Linux environment [84]. Similarly, Pinto et al. [104] created a FreeRTOS based execution environment where the trusted code is run on the ARM TrustZone as a low priority thread of an RTOS. All existing work, however, ignore the overhead associated with TEE and its impacts on hard real-time deadlines.

6.8 Summary

We tackled the challenges associated with using TEE in hard real-time systems without affecting the security and isolation features of TEE, nor requiring source code modifications.

We introduced the concept of super-TEEs, where multiple secure sections or application code that require TEE execution, are fused together to amortize TEE execution overhead while maintaining logical correctness and improving timing predictability through reduced I/O and switches between non-secure mode and TEE mode of execution. To schedule super-TEEs, we presented a sufficient condition for schedulability for uniprocessors and a fixed-priority task assignment and scheduling algorithm (ct-RM) for multicore systems. Experimental results on a real hardware platform show that ct-RM improves the usable utilization over RM-FF by up to 27% and, 12% on average, and are confirmed by simulations.

Chapter 7

A TEE-Aware DAG Scheduling Framework for Hard Real-Time Applications

The constant need for performance improvement in real-time systems has led to the design of a wide range of multicore hardware-software frameworks, with the aim to utilize the maximum scope for performance within the given time constraints [40, 65]. Standardized parallel application frameworks, for e.g., Pthreads [25] and OpenMP [29], provide a way to integrate multithreading and multiprocessing design practices in real-time applications. Among several parallel task models [74, 116], directed acyclic graph (DAG) based real-time task model [17] has been shown to successfully capture the irregular patterns of execution of applications, which use OpenMP or Ptherads to parallelize tasks, while providing highly deterministic timing predictability on multicore systems [117, 124, 125, 127]. Previous work in DAG task scheduling have also shown that the problem of achieving least schedule length, and thereby least application makespan is an NP-complete problem [57, 123] making it a suitable candidate for deriving timing predictability in complex real-time systems. However, despite the conservative worst-case response time analyses of DAG based dynamic task

scheduling [124], multithreading libraries, especially OpenMP, are not used as the industry standard to develop hard real-time applications.

Safety-critical and mission-critical real-time systems, for e.g., automotive and aerospace industry, require both privacy/isolation, and deadline guarantees. They strictly follow hard real-time deadlines through a highly predictable fixed task configuration, and a static allocation policy. While many existing approaches rely on expensive, custom-built hardware with long time-to-market or time-to-deployment to achieve enhanced security, trusted execution environment (TEE) provides an inexpensive and timely alternative for security through platform virtualization by leveraging hardware security extensions. We have already discussed the variety of security benefits with TEE, which is the most popular industry standard solution for embedded systems. However, as reported in Chapter 4, utilizing TEE creates multiple challenges from a real-time perspective, including additional execution time overheads, non-deterministic execution, and weakening timing predictability [85]. Specifically, each instance of TEE execution (in ARM TrustZone) is initiated by a setup phase and exits through a destroy phase. Since TEE leverages architecture-specific secure monitor calls (SMC) to realize these phases, the time overhead associated with TEE execution, including setting up and tearing down a TEE session, requires $18{,}500\,\mu s$ on a Raspberry Pi 3B (Rpi 3B) (Table 5.2).

The work in [92] studied the existing parallelism in trusted execution, and proposes a new task model, Split-TEE, along with a heuristic real-time task scheduling algorithm. Specifically, Split-TEE takes advantage of the independent execution context (e.g., the setup phase) during secure execution within TEE and schedules it in parallel with the non-secure execution contexts to achieve reduced application makespan, and thereby maximize its performance. However, such a heuristic scheduling algorithm has the following disadvantages; (1) it utilizes a dynamic task assignment and scheduling policy which does not comply with

the standard application configuration for validation and testing of safety-critical systems, and, (2) it does not show major improvement over existing dynamic real-time task scheduling policies [16, 18]. Conversely, Melani et. al. [88] proposes two static DAG task scheduling algorithms for OpenMP-based applications, which comply with the restrictive predictability requirements of multicore safety-critical systems, while exploiting the available parallel performance opportunities. However, these solutions are either computationally expensive or sub-optimal heuristic solutions. Moreover, they fail to take advantage of the parallelism associated with tasks which require trusted execution.

7.1 Contribution

In this work, we focus on TEE supported by ARM TrustZone since it is a popular industry standard used in smartphones and other resource constrained embedded mobile devices [95]. Our objective is to leverage the split-TEE task model to present a DAG-based static task assignment and real-time scheduling framework which complies with the industry standards used in safety-critical real-time scheduling. However, the main challenges associated with our solution include (1) amortizing the time overhead associated with SMCs, (2) reduce the overall execution overhead of trusted execution to maximize performance, and, (3) improve the temporal predictability of the system. We tackle these challenges by detecting independent segments of the code which must run in TEE, and scheduling them to run in parallel with non-secure code sections, resulting in improve application parallelism in multicore systems. Since our work focuses on optimizing trusted executions for use in hard real-time systems without changing the underlying TEE implementation, we do not examine the security benefits and drawbacks of TEE. Instead, readers are referred to existing work [56, 62, 130]. Our main contributions are as follows.

1. We present a general DAG-based real-time task model, which leverages the split-TEE task model, to capture the parallelism associated with trusted execution of hard real-time applications.

2. We develop a static task assignment and fixed priority real-time scheduling algorithm called T-DAG to schedule the tasks with trusted execution requirements on a multicore platform. In addition, we derive a sufficient condition for schedulability of the proposed T-DAG algorithm.

3. We show that our approach never performs worse, and in fact often outperforms the widely used existing state-of-the-art list scheduling techniques for multicore systems. We extensively validate our approach and assess its performance in a custom-built simulated environment.

7.2 System Model

We consider a DAG-based real-time task graph G = (V, E, T) scheduled on a homogeneous multicore system with p ∈ P cores by a work-conserving scheduling algorithm, where an eligible real-time task must be scheduled for execution if there is at least one ready core. V, E are the set of vertices and edges of the task graph respectively. Each vertex v ∈ V represents an individual task with a sequential code segment in G. Henceforth, we will use the terms vertex and task interchangeably, unless otherwise specified. Each edge (u, v) ∈ E represents the precedence relation between tasks u and v. Each task v ∈ V has its worst-case execution time (WCET) represented by a weight C_v associated with each vertex v. T is the period of execution for the overall task graph G. T is lower bounded by the latest finish time of the end vertex v_e, which is defined later. Unless otherwise specified, we will assume that task deadline D equals task period T. We also assume that each edge (u, v) ∈ E of

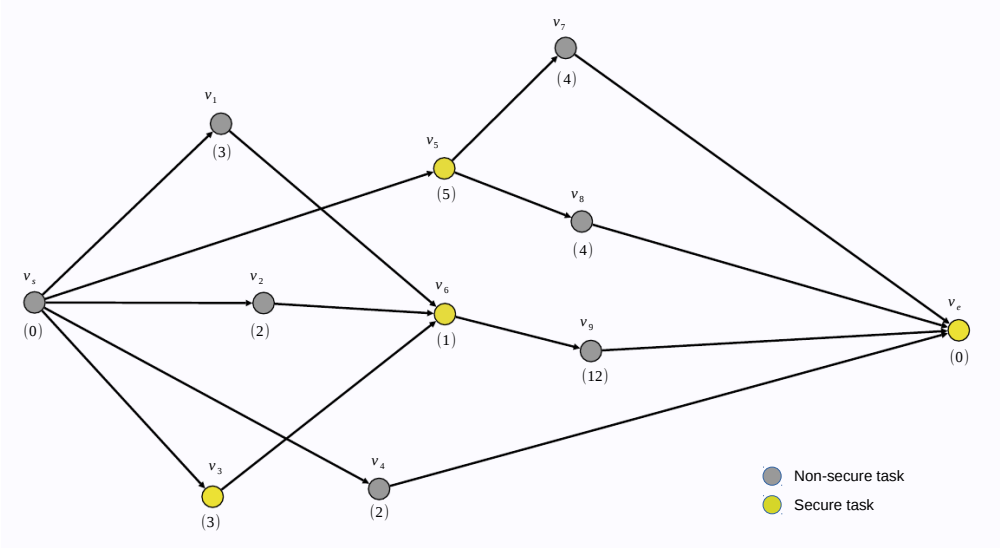


Figure 7.1: An example DAG-based real-time task graph. v_s is the source vertex and v_e is the end vertex. The WCETs of each vertex is given within brackets. The complete path $\pi = \{v_s, v_1, v_6, v_9, v_e\}$ is a critical path of the DAG-based task graph. Therefore, the values span(G) and work(G) for the DAG are 16 and 36 respectively.

the DAG-based task graph G does not have an associated weight. Next, we will revisit the properties of a DAG-based task graph (shown in figure 7.1) that will help in our response time analysis later.

Property 7.1. If there is an edge $(u, v) \in E$, u is the predecessor of v. Similarly, v is the successor of u. We will use pre(u) and suc(u) to denote the set of predecessors and successors of a task u in the task graph G.

Property 7.2. A task graph G has a unique start vertex v_s (which has no predecessor) and a unique end vertex v_e (which has no successor). The WCETs of both v_s and v_e are assigned as zero, i.e., $C_s = 0$ and $C_e = 0$ respectively.

Property 7.3. We use $\pi \in G$ to denote a path in G. A path $\pi = \{v_s, v_2, v_3, \ldots, v_e\}$ is a complete path if its first vertex is the source vertex v_s of G and last vertex is the sink vertex v_e. The length of a path π the sum total of the WCETs of all the vertices in the path, and

is denoted by span(π). The length of a critical path (π_{cp}) in the task graph G is given by

$$\text{span(G)} = \max_{\pi \in \text{G}} \left\{ \sum_{\text{v} \in \pi_{\text{cp}}} C_v \right\} \tag{7.1}$$

Property 7.4. We use work(G) to represent the cumulative WCET of all the tasks in G, and quantify it as $\text{work(G)} = \sum_{\text{u} \in \text{V}} \text{C}_\text{u}$.

Definition 7.1. The response time bound R(G) for a DAG-based task graph G scheduled on a P-core platform [54] is given by

$$R(G) \leq span(G) + \frac{work(G) - span(G)}{P} \tag{7.2}$$

By the definition of critical path, we know that the predecessor $\text{v}_{\text{i}-1}$ of a task v_i in the critical path must finish execution by the start time $\text{T}_{\text{v}_\text{i}}$ of v_i, and is bounded by $\text{T}_{\text{v}_\text{i}} \geq \text{T}_{\text{v}_{\text{i}-1}} + \text{C}_{\text{v}_{\text{i}-1}}$. Note that in a critical path $\pi_{\text{cp}} \in \text{G}$, no vertex has more than one predecessor ($\mid \text{pre}(\text{v}_\text{i}) \mid = 1$). Also, by work conserving principle, when v_i cannot execute at its earliest start time $\text{T}^{\text{s}}_{\text{v}_\text{i}}$ ($= \text{T}_{\text{v}_{\text{i}-1}} + \text{C}_{\text{v}_{\text{i}-1}}$), each core must be busy executing a vertex v_j not in the critical path (π_{cp}). That is, a vertex $\text{v}_\text{j} \in \text{V}^* \mid \text{V}^* = \{\text{V} \cap \pi_{\text{cp}}\}$ must be contending with the execution of v_i. Therefore, intuitively, since cumulative WCET of vertices $\text{v}_\text{j} \in \text{V}^*$ of task graph G that are not on the critical path is at most work(G) $-$ span(G) and, the number of available cores for execution is P, the total time during which the vertices on critical path can be delayed for execution, in the worst-case scenario, is bounded by $\frac{work(G) - span(G)}{P}$.

7.2.1 Task Model

Our DAG-based real-time task graph G consists of tasks which can run in normal environment (non-secure tasks), and other tasks which needs to be run inside TEE (secure tasks).

In figure 7.1, the secure tasks are depicted in yellow, while the non-secure tasks are depicted otherwise. Each vertex, i.e., a task in G, can be modeled as a real-time task as shown in Figure 7.2. For a task v_i that needs to run inside a TEE, two normal computation segments (C_i^q $q = 1, 2$) are interleaved by a single trusted execution segment[1] (υ_i). For a task v_i that does not require TEE, $\upsilon_i = C_i^2 = 0$. The upper bound on the combined WCET of a task v_i is given by $C_{v_i} = \sum_{q=1}^{2} C_i^q + \upsilon_i$. Similarly, the upper bound on the trusted execution section is given by $\upsilon_i = \upsilon_i^{\mathrm{s}} + \upsilon_i^{\mathrm{t}} + \upsilon_i^{\mathrm{d}}$, where υ_i^s is the SMC_setup time, υ_i^d is the SMC_destroy time, and υ_i^t is the time taken to perform actual trusted computation inside TEE. We assume a set of homogeneous cores P, where each core $p \in$ P can switch between secure (TEE) and non-secure (non-trusted OS) mode of operation using SMCs.

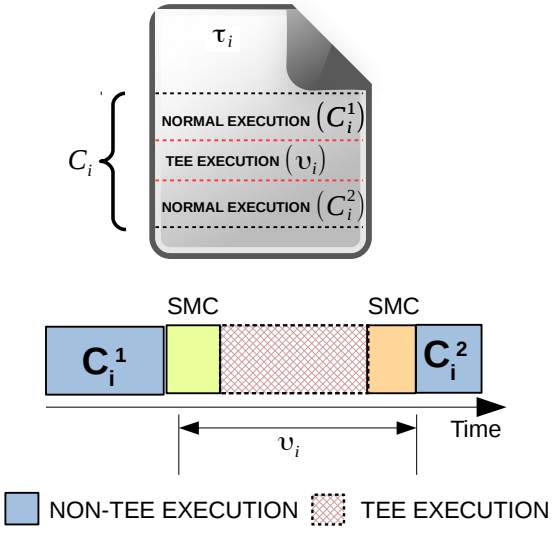


Figure 7.2: Our real-time task model. For a task v_i without TEE requirement, $\upsilon_i = C_i^2 = 0$.

[1]Though we consider a single trusted execution interval for simplicity, it is not advisable to have multiple TEE execution sections interspersed with non-secure computation segments within a secure task, since each trusted execution has high time overhead (Table 5.2).

7.2.2 Modeling DAG-based Parallelism for Secure Execution

A task that needs to run inside a TEE, i.e., a secure task, incurs additional execution overhead, which can be attributed to the SMCs, resulting in increased overall makespan of the DAG-based task graph G. Previously, in Chapters 5-6, we addressed this by either (1) heuristically exploiting the data dependency between secure and non-secure tasks to improve the overall usable utilization of a system, or, (2) exploiting the TEE's design-specific redundancy to optimize the execution of secure tasks within the TEE environment, and improve both system performance and predictability. In this work, we tackle the problem of execution overhead by exploiting the existing parallelism in trusted execution within TEE, while maintaining the required level of system predictability for safety-critical hard real-time systems. For that, we leverage the split-TEE task model [92] which makes the following observations; (1) the TEE setup phase is independent of the rest of the TEE execution (actual trusted computation duration inside TEE and the TEE destroy phase), and, (2) TEE setup phase can be scheduled ahead of time as long as the execution context and the application's logical correctness remains intact.

We capture the said parallelism by presenting a secure task model which fits into our DAG-based real-time task graph G. We transform our DAG-based real-time task graph by decoupling the TEE setup phase from the rest of the secure task $v_i \in V$ and represent it as a separate vertex v_i^s in the DAG as shown in Figure 7.3. The weight associated with each v_i^s is set to the upper bound of υ_i^s, while the rest of the secure task, represented by vertex v_i^t, is assigned the weight equal to $\upsilon_i^t + \upsilon_i^d$. We now introduce a list of new features for our proposed DAG-based task model.

Property 7.5. The maximum interference on a task v in the critical path (π_{cp}) is defined as the difference between its actual start time T_v and its earliest start time T_v^s.

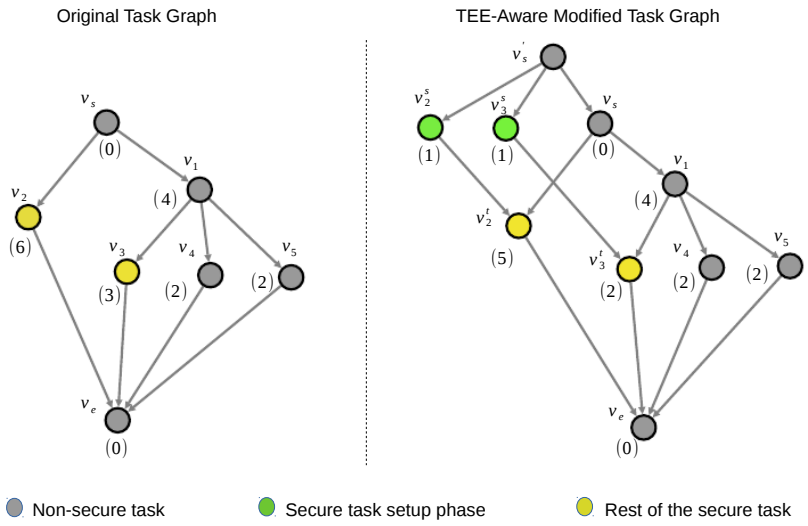


Figure 7.3: An example DAG-based TEE-aware real-time task graph, formed by transforming a normal DAG-based real-time task graph. The vertices colored green represent the setup time phases of the secure tasks, while the vertices colored yellow represent the rest of the secure tasks. v_s is the source vertex and v_e is the end vertex. A pseudo-source vertex v'_s is used to comply with Property 7.2. The complete path $\pi = \{v'_s, v_s, v_1, v_3, v_e\}$ is a critical path of the original DAG.

Property 7.6. For each vertex $v \in V$ in a DAG task G, par(v) denotes the set of vertices that are neither predecessors nor successors of v.

Property 7.7. The set of vertices in the DAG-based task graph G that are not on the critical path π_{cp} but can actually interfere with vertices in π_{cp} is denoted by $vcp(\pi, G)$. Therefore,

$$vcp(\pi_{cp}, G) = \bigcup_{v_i \in \pi_{cp}} par(v_i) \tag{7.3}$$

Property 7.8. The level of a vertex v_i in a DAG is defined as the longest path from v_i to the source vertex (v_s) and, is denoted by $L(v_i)$.

7.3 T-DAG: Static Scheduling Algorithm

Transforming a normal DAG-based real-time task graph G to our proposed TEE-aware DAG-based task model results in increase in $\text{vcp}(\pi, \text{G})$. From Figure 7.3, we can see that vertex v_2, which represents a TEE task can be split into two separate vertices v_2^{s} and v_2^{t} with updated weights. Thereby, v_2^{s} can run in parallel with v_s, which is the first node of the critical path π_{cp} in the original task graph G. By definition, since v_2^{s} is neither a predecessor nor a successor of v_1, we can include v_2^{s} in $\text{par}(\text{v}_1)$ and by extension, in $\text{vcp}(\pi_{\text{cp}}, \text{G})$. This also means that nodes v_2^{s} can cause interference to the execution of any task that is in the critical path π_{cp} due to resource contention.

We leverage the list scheduling algorithm [103, 108] to present a static task assignment and scheduling framework, T-DAG, to schedule the TEE-aware DAG task graph on a safety-critical hard real-time system consisting of P processors, as shown in Algorithm 7.1. Our task prioritization phase is shown in lines 2-9. We first assign the highest priority to vertices which represent TEE setup phases. Next, we assign medium priority to the secure tasks, followed by lowest priority to non-secure tasks. The task-to-core assignment shown in lines 11-26 is as follows. We first assign the $\text{v}_\text{i}^{\text{s}}$ tasks (which represents TEE setup phase) to the available processors. If a secure task (v_i) is ready for execution, it is assigned to the same core which ran its corresponding $\text{v}_\text{i}^{\text{s}}$. The ready normal sub-tasks are assigned to the remaining cores according to their decreasing priority.

We assume that the number of secure tasks are much fewer compared to non-secure tasks. We want to limit the number of secure tasks to minimum because our experiment on real-world embedded platforms show that secure mode of execution (1) incurs very high overhead and, (2) offers limited hardware and software support for trusted applications (for e.g., stripped C standard library, fewer peripherals etc.). Therefore, it is wise to limit the proprietary

information in a real-world mission-critical system to a few lines of code. We further assume that no TEE-aware DAG task graph G can start with a secure task. For instance, the vertex v_s of the original DAG-based task graph in Figure 7.3 has to represent a non-secure task.

Algorithm 7.1 T-DAG Task-to-Core Mapping

```
1: function Priority_Assignment(V, E, G, P)
2:     ▷ Rank task tuples
3:     for v_i ∈ V do
4:         if v_i is v_i^s then
5:             v_i ← HP                          ▷ HP : high priority
6:         else if v_i is v_i^t then
7:             v_i ← MP                          ▷ MP : Medium priority
8:         else
9:             v_i ← LP                          ▷ LP : Low priority
10: function Task_Assignment(V, E, G, P)
11:     ▷ TEE-aware DAG scheduling
12:     RL ← v_s                                  ▷ RL: ready list; Source vertex queued.
13:     Sort_Task_Priority(RL)
14:     while RL¬{∅} do
15:         v_j ← Pop(RL)
16:         if v_j == v_j^s then
17:             assign v_j to new p_i
18:             SP(v_j^s) ← p_i                   ▷ Save which processor gets which v_j^s
19:         else if v_j == v_j^t then
20:             Find p_x | SP(v_j^s) = p_x
21:             assign τ_j to p_x
22:         else
23:             assign v_j to idle p_i
24:         if ¬SP(v_j) then
25:             Push(RL)
26:         Sort_Task_Priority(RL)
27: function main(V, E, G, P)
28:     Priority_Assignment(V, E, G, P)
29:     Task_Assignment(V, E, G, P)
30:     EDF_Schedule()
31:     return
```

7.4 Schedulability Analysis of T-DAG

The main objective for exploiting the existing parallelism in trusted execution is to reduce the overall makespan, and thereby save CPU cycles every time an instance of DAG-based task graph is scheduled. While decoupling the TEE setup phases from the rest of the secure tasks and running them in parallel with other non-secure tasks may reduce the collective makespan of the task graph, such a modification does change the collective makespan of the critical path π_{cp} of the task graph under consideration, by increasing the total interference due to resource contention between tasks in π_{cp} and the newly created vertices representing the TEE setup phase. We present a schedulability analysis to determine whether such a transformation is in fact feasible, i.e, the newly created vertices will not cause an increase in the worst-case response time of the overall task graph. Based on the proposed approach as shown in Algorithm 7.1, we will report the following observations. We will divide the response time analysis of the DAG-based real-time task into two distinct cases. We will first derive the response-time analysis for a DAG task G when scheduled on a platform with unbounded resources, i.e., infinite number of cores.

Corollary 7.1. The worst-case response time bound of a DAG-based task graph G on a system with ∞ core count is

$$R(G) \leq span(G) \tag{7.4}$$

Proof. Using the Equation 7.2, when the value of P tends to ∞, the response time bound of such a system reduces down to $\text{R(G)} \leq \text{span(G)}$. ■

We will now derive a condition which will upper bound the response-time for a DAG task graph G when scheduled on a platform with bounded number of cores.

Lemma 7.1. For a system where available number of cores $\text{p} \in \text{P}$ in the system is equal to the

number of secure tasks, the maximum interference on the critical path π_{cp} of a TEE-aware DAG task graph G due to a set of newly created vertices $\{v_i^s\}$, representing TEE setup phase, is given by

$$\sum\{\mathrm{C}_{\mathrm{v_i^s}}\} \mid \mathrm{v_i^s} \in \mathrm{vcp}(\pi_{\mathrm{cp}}, \mathrm{G}), \tag{7.5}$$

where number of available scheduling cores is equal to number of TEE tasks.

Proof. Consider that the available number of cores in the system is equal to the number of secure tasks. Therefore, we can schedule each newly created vertex $\mathrm{v_i^s}$ to an individual core for execution without any interference due to resource contention. The $\mathrm{L}(\mathrm{v_i^s})$ of each vertex $\mathrm{v_i^s}$ is same, and equal to $\mathrm{L}(\mathrm{v_s})$, where $\mathrm{v_s}$ denotes the first vertex of the critical path of a DAG-based task graph before being transformed to TEE-aware DAG task graph. Therefore, all $\mathrm{v_i^s}$ tasks are scheduled in parallel with $\mathrm{v_i^s}$, and run immediately due to their assigned highest priority on separate cores. The earliest time when a non-secure task in the critical path with non-zero weight can start executing on any core $\mathrm{p} \in \mathrm{P}$, is when the vertex $\mathrm{v_i^s}$ with shortest WCET completes execution on that core p. Therefore, the maximum delay in start time of the critical path π_{cp} is $min\{C_{v_i^s}\}$. ■

Theorem 7.1. The minimum number of scheduling processors $\mathrm{P^{max}}$ required by a TEE-aware DAG task graph to obtain best-case response time is given by,

$$\mathrm{P^{max}} = \max\{(P^{'} + 1), N\}, \tag{7.6}$$

where $\mathrm{P}^{'}$ is the number of secure tasks, and N is the minimum number of cores required to obtain the best-case response time of a DAG task graph before our proposed transformation.

Proof. This follows directly from Lemma 7.1. Let us consider the TEE-aware DAG-based task graph as shown in Figure 7.3. Since the newly created vertex $\mathrm{v_i^s} \in \mathrm{vcp}(\pi_{\mathrm{cp}}, \mathrm{G})$, repre-

senting TEE setup phase, may cause interference to a task $\mathrm{v_i} \in \pi_{\mathrm{cp}}$, we need to assign each vertex $\mathrm{v_i^s}$ to a separate core. Similarly, since $\mathrm{L(v_i^s)} \leq \mathrm{L(v_i)}$, the rest of the DAG starting with vertex $\mathrm{v_j} \in \mathrm{suc(v_i)}$ can already be scheduled on N processors to obtain its best-case response time. Therefore, we need at least $\max\{(\mathrm{P}^{'}+1), \mathrm{N}\}$ cores to obtain the best-case response time of the TEE-aware DAG task graph under consideration. ■

Theorem 7.2. A TEE-aware DAG task graph scheduled on $\mathrm{P^{max}}$ processors will have at most the same worst-case response time as a DAG task graph, before transformation, scheduled on P processors if,

$$min\{C_{v_i^s}\} - \frac{\sum\limits_{v_i \in v_i^s} v_i}{P^{max}} \leq \frac{work(G) - span(G)}{P} - \frac{work(G) - span(G)}{P^{max}}, \tag{7.7}$$

where $\mathrm{P} \geq \mathrm{P^{max}}$, the minimum number of scheduling processors required by a TEE-aware DAG task graph to obtain its best-case response time.

Proof. Lemma 7.1 shows that the maximum delay in the execution of critical path in an P-core system is given by

$$\min\{C_{v_i^s}\} \mid v_i^s \in vcp(\pi, G), \tag{7.8}$$

Similarly, we know that a vertex $\mathrm{v_i^s}$, representing TEE setup phase, can be included as part of $vcp(\pi, G)$. Therefore, the maximum amount of parallel work done by P processors for running all $\mathrm{v_i^s}$ tasks is given by

$$\frac{\sum\limits_{v_i \in v_i^s} v_i}{P} \tag{7.9}$$

In order to maintain the worst-case response time bound for the original DAG task graph,

before transformation, we must have

$$span(G) + \min\{C_{v_i^s}\} + \frac{work(G) - span(G)}{P^{max}} - \frac{\sum\limits_{v_i \in v_i^s} v_i}{P^{max}} \leq R(G)$$

where

$$R(G) \leq span(G) + \frac{work(G) - span(G)}{P}$$

Solving the above inequality, we get

$$\min\{C_{v_i^s}\} - \frac{\sum\limits_{v_i \in v_i^s} v_i}{P^{max}} \leq \frac{work(G) - span(G)}{P} - \frac{work(G) - span(G)}{P^{max}}$$

■

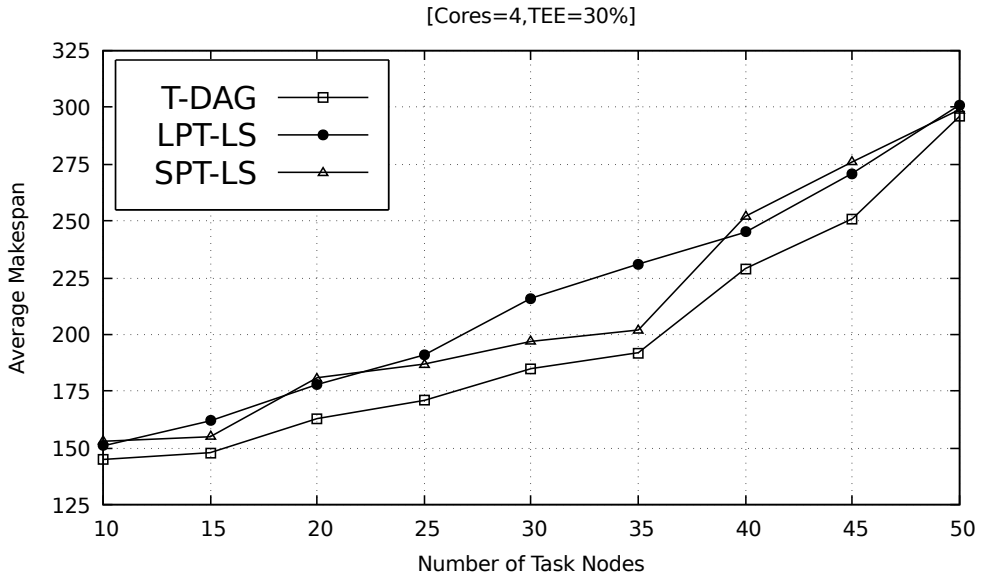


Figure 7.4: Simulation results showing the average application makespan as a function of scaling task nodes with fixed core count (= 4) and secure tasks (= 30%).

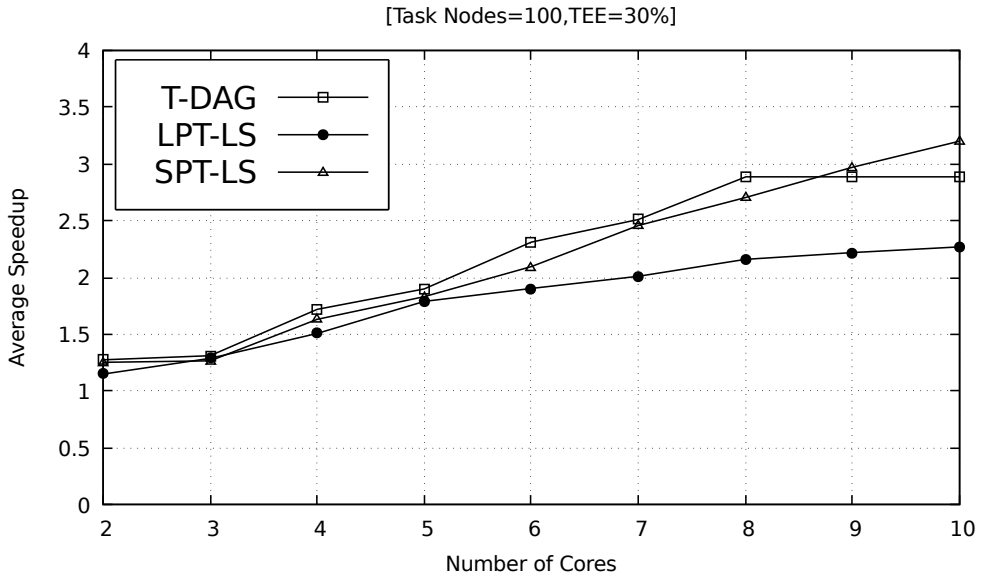


Figure 7.5: Simulation results showing the average system speedup as a function of scaling core count with fixed number of task nodes (= 100) and secure tasks (= 30%).

7.5 Evaluation

In this section, we evaluate the benefits of our proposed approach by scheduling synthetic DAG-based real-time task graphs on a custom-built simulated environment.

7.5.1 Simulation Setup

We validate our proposed task assignment and scheduling framework (T-DAG) and assess its real-time performance against the following two high performing heuristic scheduling strategies for DAG based list scheduling (LS) algorithms for multicore systems [103, 108].

1. Longest processing time (LPT-LS), where the task with the higher WCET is assigned a higher priority.

2. Shortest processing time (SPT-LS), where the task with the lower WCET is assigned

a lower priority.

Our simulation environment consists of randomly generated task sets through our custom-built task set generator. We generate DAG task graphs over a range of number of vertices (10%, 15%, 20%, 25%,. . ., 100%). For a fixed number of vertices, we generate 100 DAG task graphs, each of which has secure tasks ranging between $30\% - 50\%$ of the total number of tasks. Each simulation run tests the feasibility of the generated task sets using the steps listed in Algorithm 7.1

7.5.2 Simulation Metrics

The main objective of list scheduling algorithm is to minimize the schedule length of an application program, which is also know as the makespan of an application. We use this metric along with system speedup as the performance metrics for T-DAG over the other two scheduling policies, namely LPT-LS and SPT-LS. We define speedup as overall computation time of a DAG (i.e., the makespan) when running on a single processing core, over the reported overall computation time of a DAG (i.e., the makespan) on a multicore system. Therefore, speedup (S) of a given DAG task G is given by,

$$S = \frac{work(G)}{makespan(G)} \tag{7.10}$$

7.5.3 Simulation Results

Figure 7.4 reports the performance of T-DAG against LPT-LS and SPT-LS by comparing the average makespan with varying number of tasks. For the reported simulation results, we vary the number of tasks of the DAG task graphs while keeping the number of secure

tasks fixed at 30%, while running on a simulated multicore platform consisting of 4 cores. Our results indicate that T-DAG outperforms LPT-LS (SPT-LS) algorithm with an average makespan improvement of 9%(6%) and up to 17%(15%).

Figure 7.5 reports the performance of T-DAG against LPT-LS and SPT-LS in terms of the average system speedup with scaling number of cores. For the reported simulation results, we fix the number of tasks of the DAG task graphs to 100, and the total number of secure tasks is fixed at 30%, while running on a simulated multicore platform with scaling core count ranging between $(1, 2, 3, \ldots, 10)$. Our results indicate that T-DAG outperforms LPT and SPT list scheduling algorithms with an average speedup improvement of 28% and up to 25% respectively. It also achieves its best-case response times at lower core count, i.e, 8, as compared to the other two existing DAG scheduling algorithms.

7.6 Summary

In this work, we introduce a DAG-based real-time task model, with support for trusted execution in safety-critical hard real-time systems. We present a static task assignment and scheduling framework (T-DAG) which achieves improved performance, while maintaining timing and logical correctness. We also derive a sufficient condition for schedulability of T-DAG. We show that our approach never performs worse, and in fact often outperforms the widely used existing list scheduling algorithms for multicore systems in a custom-built simulated environment.

Chapter 8

Conclusions

The major paradigms of hardware-software co-design of resource constrained real-time systems has been an area of active research in the past several years. In this dissertation, we aim to explore the gap in research for performance-aware and energy-aware resource management in embedded systems, especially in safety-critical and mission-critical hard real-time systems. To address the trade-off between performance and power in design decisions on resource constrained mobile embedded systems, in Chapter 3, we propose an integrated energy management solution, that aims to minimize the system-wide energy consumption with negligible effect on QoS in smartphones. To tackle the trade-off between performance and timing predictability in design decisions on safety-critical hard real-time systems, we first present a global heuristic multicore scheduling framework, which enables the use of trusted execution on real-time systems with improved predictability in Chapter 5. To strengthen the predictability of scheduling hard real-time tasks on secure embedded processors, with improved usable system utilization, we present a novel task model, along with a task assignment and scheduling algorithm for multicore scheduling in Chapter 6. Finally, in Chapter 7, we introduce a TEE-aware DAG static task assignment and multicore scheduling algorithm

that aims to optimize the worst-case response time of parallel safety-critical applications.

8.1 Future Research Directions

We now introduce two broad research problems that can leverage the works presented in this dissertation to meet the existing design challenges of resource constrained real-time systems.

8.1.1 Extending the Energy Management Framework for Mobile Embedded Systems

In Chapter 3, we proposed an effective joint energy management solution, which considers the varying resource usage pattern of an application over time, and presenting significant power saving opportunities on processors and memory subsystem, without sacrificing application QoS. We established that optimizing the energy consumption of each component independently does not always result in minimum system-wide energy consumption, especially as the components must often inter-operate. For instance, the overall system load is often used to adjust core voltage and frequency settings to optimize application QoS, resulting in sub-optimal energy solutions. However, our solution ignores peripheral device usage (for ex. GPU, DMA, IP co-processors, display, etc.) which have been shown to be application-specific [109] too. Since the power consumption of these peripherals are now comparable to that of processors [119], effective energy-aware designs must not only consider the power consumption due to processor cores and memory subsystem but also due to peripherals, especially since all the peripheral devices are integrated within an SoC with a high functional dependency between various components.

8.1.2 Extending the Trusted Execution Framework for Energy Constrained Safety-Critical Systems

In Chapter 3, proposed a energy management framework to address the challenges of resource-constrained embedded systems. Similarly, in Chapters 5-7, we explored design challenges in TEE-enabled safety-critical systems. We can aim to solve the design overhead associated with TEE execution by extending our proposed energy management techniques to secure execution contexts. We know that each TEE execution is divided into separate setup phase, execution phase, and, destroy phase. Also, setup and destroy phases perform code and/or data copy from/to secure to/from non-secure sections of the memory. Therefore, we can thus identify distinct compute-intensive and memory-intensive phases of TEE execution. Thus, we can minimize the time overhead associated with TEE execution by voting for the best voltage and frequency configuration, which can reduce execution overhead, thereby improving real-time performance while minimizing the system-wide energy consumption.

Bibliography

[1] GlobalPlatform Device Technology TEE Client API Specification. https://www.globalplatform.org/mediaguidetee.asp. Accessed: 2017-10-05.

[2] OP-TEE (Open Portable Trusted Execution Environment). https://www.op-tee.org/. Accessed: 2018-05-27.

[3] Pixhawk Autopilot(2005). http://pixhawk.org/. Accessed: 2017-10-06.

[4] I Abubakar, SN Khalid, MW Mustafa, Hussain Shareef, and M Mustapha. Application of load monitoring in appliances' energy management – a review. Renewable and Sustainable Energy Reviews, 67:235–245, 2017.

[5] Enas Ahmad and Basem Shihada. Green smartphone GPUs: Optimizing energy consumption using GPUFreq scaling governors. In International conference on Wireless and Mobile Computing, Networking and Communications (WiMob), pages 740–747, 2015.

[6] Quazi N Ahmed and Susan V Vrbsky. Maintaining security in firm real-time database systems. In Computer Security Applications Conference, 1998. Proceedings. 14th Annual, pages 83–90. IEEE, 1998.

[7] Android Developers. Android SDK. http://developer.android.com/sdk/ndk/index,html, 2013.

[8] Android Developers. Monkey Command-Line Emulator. https://developer.android.com/studio/test/monkey, (2018).

[9] Android Developers. Background Execution Limits. https://developer.android.com/about/versions/oreo/background.html, (2018).

[10] Android Developers. Background Location Limits. https://developer.android.com/about/versions/oreo/background-location-limits.html, (2018).

[11] Android Developers. Android 8.0 Behavior Changes. https://developer.android.com/about/versions/oreo/android-8.0-changes.html, (2018).

[12] ARM. Security technology building a secure system using trustzone technology (white paper). ARM Limited, 2009.

[13] Theodore P Baker and Alan Shaw. The cyclic executive model and ada. Real-Time Systems, 1(1):7–25, 1989.

[14] T.P. Baker and S.K. Baruah. An analysis of global EDF schedulability for arbitrary-deadline sporadic task systems. Real-Time Systems, 43(1):3–24, September 2009.

[15] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang. On the competitiveness of on-line real-time task scheduling. Real-Time Systems, 4(2):125–144, May 1992.

[16] Sanjoy Baruah and Theodore Baker. Schedulability analysis of global edf. Real-Time Systems, 38(3):223–235, 2008.

[17] Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. A generalized parallel task model for recurrent real-time processes. In 2012 IEEE 33rd Real-Time Systems Symposium, pages 63–72. IEEE, 2012.

[18] Sanjoy K Baruah and John Carpenter. Multiprocessor fixed-priority scheduling with restricted interprocessor migrations. Journal of Embedded Computing, 1(2):169–178, 2005.

[19] S.K. Baruah. Scheduling periodic tasks on uniform multiprocessors. In Proc. Euromicro Conf. Real-Time Systems, pages 7–13, June 2000.

[20] Stefano Bertozzi, Andrea Acquaviva, Davide Bertozzi, and Antonio Poggiali. Supporting task migration in multi-processor systems-on-chip: a feasibility study. In Proceedings of the conference on Design, automation and test in Europe: Proceedings, pages 15–20. European Design and Automation Association, 2006.

[21] Pamela Thays Bezerra, Leandro AB Araujo, Giovane Boaviagem Ribeiro, Antonio Correia de Sa Barreto Neto, Abel Guilhermino Silva-Filho, Clauirton A Siebra, Fabio QB da Silva, Andre LM Santos, Angelica Mascaro, and Paulo HR Costa. Dynamic frequency scaling on android platforms for energy consumption reduction. In Proceedings of the ACM workshop on Performance monitoring and measurement of heterogeneous wireless and wired networks, pages 189–196, 2013.

[22] Alessandro Biondi, Giorgio C Buttazzo, and Marko Bertogna. Schedulability analysis of hierarchical real-time systems under shared resources. IEEE Transactions on Computers, 65(5):1593–1605, 2016.

[23] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against aes. In International Workshop on Cryptographic Hardware and Embedded Systems, pages 201–215. Springer, 2006.

[24] José V Busquets-Mataix, Juan José Serrano, Rafael Ors, Pedro Gil, and A Wellings. Using harmonic task-sets to increase the schedulable utilization of cache-based preemptive real-time systems. In Proceedings of 3rd International Workshop on Real-Time Computing Systems and Applications, pages 195–202. IEEE, 1996.

[25] Dick Buttlar, Jacqueline Farrell, and Bradford Nichols. PThreads Programming: a POSIX Standard for better multiprocessing. " O'Reilly Media, Inc.", 1996.

[26] Aaron Carroll and Gernot Heiser. Unifying DVFS and offlining in mobile multicores. In Real-Time and Embedded Technology and Applications Symposium (RTAS), pages 287–296, 2014.

[27] Ann Cavoukian. Privacy and drones: Unmanned aerial vehicles. Information and Privacy Commissioner of Ontario, Canada Ontario, Canada, 2012.

[28] David Carroll Challener and David Robert Safford. Encrypted file system using tcpa, March 11 2008. US Patent 7,343,493.

[29] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. Parallel programming in OpenMP. Morgan kaufmann, 2001.

[30] Nitin Chaudhary, Thummala Pallavi, et al. Bus bandwidth monitoring, prediction and control. In International conference on Advances in Computing, Communications and Informatics (ICACCI), pages 1152–1158, 2015.

[31] Jian-Jia Chen and Cong Liu. Fixed-relative-deadline scheduling of hard real-time tasks with self-suspensions. In Real-Time Systems Symposium (RTSS), 2014 IEEE, pages 149–160. IEEE, 2014.

[32] Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil Audsley, et al.

Many suspensions, many problems: A review of self-suspending tasks in real-time systems. Technical Report 854, Faculty of Informatik, TU Dortmund, 2016.

[33] Xi Chen, Chi Xu, and Robert P Dick. Memory access aware on-line voltage control for performance and energy optimization. In Proceedings of the International Conference on Computer-Aided Design, pages 365–372, 2010.

[34] Xiang Chen, Jiachen Mao, Jiafei Gao, Kent W Nixon, and Yiran Chen. MORPh: mobile OLED-friendly recording and playback system for low power video streaming. In Proceedings of the Annual Design Automation Conference, page 153, 2016.

[35] H. Cho, B. Ravindran, and E.D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In Proc. Real-Time Systems Symp., pages 101–110, December 2006.

[36] N Corteggiani, G Camurati, and A Francillon. Inception: System-wide security testing of real-world embedded systems software. In 27th USENIX Security Symposium (USENIX Security 18). USENIX Association, 2018.

[37] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal risc extensions for isolated execution. IACR Cryptology ePrint Archive, 2015:564, 2015.

[38] Saad M Darwish, Shawkat K Guirguis, and Mohamed S Zalat. Stealthy code obfuscation technique for software security. In Computer Engineering and Systems (ICCES), 2010 International Conference on, pages 93–99. IEEE, 2010.

[39] R.I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. ACM Computing Surveys, 43(4):1–44, October 2011.

[40] Benoît Dupont De Dinechin, Duco Van Amstel, Marc Poulhiès, and Guillaume Lager. Time-critical computing on a single-chip massively parallel processor. In 2014 Design,

Automation & Test in Europe Conference & Exhibition (DATE), pages 1–6. IEEE, 2014.

[41] Arnaldo Carvalho De Melo. The new linux perf tools. In Slides from Linux Kongress, volume 18, pages 1–42, 2010.

[42] U.C. Devi. Soft real-time scheduling on multiprocessors. PhD thesis, University of North Carolina at Chapel Hill, 2006.

[43] UmaMaheswari C Devi. An improved schedulability test for uniprocessor periodic task systems. In Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on, pages 23–30. IEEE, 2003.

[44] Sudarshan K Dhall and Chung Laung Liu. On a real-time scheduling problem. Operations research, 26(1):127–140, 1978.

[45] Xiaowan Dong, Sandhya Dwarkadas, and Alan L Cox. Characterization of shared library access patterns of android applications. In International Symposium on Workload Characterization (IISWC), pages 112–113, 2015.

[46] Zheng Dong and Cong Liu. Closing the loop for the selective conversion approach: a utilization-based test for hard real-time suspending task systems. In 2016 IEEE Real-Time Systems Symposium (RTSS), pages 339–350. IEEE, 2016.

[47] Lin-Tao Duan, Michael Lawo, Ingrid Rügge, and Xi Yu. Power management of smartphones based on device usage patterns. In Dynamics in Logistics, pages 197–207. Springer, 2017.

[48] N. Fisher, J. Goossens, and S. Baruah. Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. Real-Time Systems, 45(1–2):26–71, June 2010.

[49] Ian Foster. Designing and building parallel programs, volume 78. Addison Wesley Publishing Company Boston, 1995.

[50] Torsten Frenzel, Adam Lackorzynski, Alexander Warg, and Hermann Härtig. Arm trustzone as a virtualization technique in embedded systems. In Proceedings of Twelfth Real-Time Linux Workshop, Nairobi, Kenya, 2010.

[51] Yong Fu, Nicholas Kottenstette, Yingming Chen, Chenyang Lu, Xenofon D Koutsoukos, and Hongan Wang. Feedback thermal control for real-time systems. In 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium, pages 111–120. IEEE, 2010.

[52] Keke Gai, Longfei Qiu, Min Chen, Hui Zhao, and Meikang Qiu. Sa-east: security-aware efficient data transmission for its in mobile heterogeneous cloud computing. ACM Transactions on Embedded Computing Systems (TECS), 16(2):60, 2017.

[53] Farshad Ghanei, Pranav Tipnis, Kyle Marcus, Karthik Dantu, Steve Ko, and Lukasz Ziarek. Os-based resource accounting for asynchronous resource use in mobile systems. In Proceedings of the International Symposium on Low Power Electronics and Design, pages 296–301, 2016.

[54] Ronald L. Graham. Bounds on multiprocessing timing anomalies. SIAM journal on Applied Mathematics, 17(2):416–429, 1969.

[55] J Greene. Intel trusted execution technology, white paper. Online: http://www. intel. com/txt, 2012.

[56] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. Trustshadow: Secure execution of unmodified applications with arm trust-

zone. In Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, pages 488–501. ACM, 2017.

[57] Tarek Hagras and Jan Janecek. A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. In 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings., page 107. IEEE, 2004.

[58] C-C Han and H-Y Tyan. A better polynomial-time schedulability test for real-time fixed-priority scheduling algorithms. In Proceedings Real-Time Systems Symposium, pages 36–45. IEEE, 1997.

[59] Monowar Hasan, Sibin Mohan, Rakesh B Bobba, and Rodolfo Pellizzoni. Exploring opportunistic execution for integrating security into legacy hard real-time systems. In Real-Time Systems Symposium (RTSS), 2016 IEEE, pages 123–134. IEEE, 2016.

[60] Monowar Hasan, Sibin Mohan, Rodolfo Pellizzoni, and Rakesh B Bobba. A design-space exploration for allocating security tasks in multicore real-time systems. In Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018, pages 225–230. IEEE, 2018.

[61] Songtao He, Yunxin Liu, and Hucheng Zhou. Optimizing smartphone power consumption through dynamic resolution scaling. In Proceedings of the Annual International Conference on Mobile Computing and Networking, pages 27–39, 2015.

[62] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. vtz: Virtualizing arm trustzone. In In Proc. of the 26th USENIX Security Symposium, 2017.

[63] Wen-Hung Huang and Jian-Jia Chen. Self-suspension real-time tasks under fixed-

relative-deadline fixed-priority scheduling. In Proceedings of the 2016 Conference on Design, Automation & Test in Europe, pages 1078–1083. EDA Consortium, 2016.

[64] Connor Imes, David HK Kim, Martina Maggio, and Henry Hoffmann. Poet: A portable approach to minimizing energy under soft real-time constraints. In Real-Time and Embedded Technology and Applications Symposium (RTAS), pages 75–86, 2015.

[65] Texas Instruments. The 66ak2h12 keystone ii processor.

[66] Gorka Irazoqui and Xiaofei Guo. Cache side channel attack: Exploitability and countermeasures. Black Hat Asia, 2017, 2017.

[67] K. Jiang, A. Lifa, P. Eles, Z. Peng, and W. Jiang. Energy-aware design of secure multi-mode real-time embedded systems with FPGA co-processors. In Proc. Int. Conf. Real-Time Networks and Systems, pages 109–118, October 2013.

[68] Ke Jiang, Adrian Lifa, Petru Eles, Zebo Peng, and Wei Jiang. Energy-aware design of secure multi-mode real-time embedded systems with fpga co-processors. In Proceedings of the 21st International conference on Real-Time Networks and Systems, pages 109–118. ACM, 2013.

[69] Wonwoo Jung, Kwanghwan Kim, and Hojung Cha. Userscope: A fine-grained framework for collecting energy-related smartphone user contexts. In International Conference on Parallel and Distributed Systems (ICPADS), pages 158–165, 2013.

[70] Uri Kanonov and Avishai Wool. Secure containers in android: the samsung knox case study. In Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices, pages 3–12. ACM, 2016.

[71] Young Geun Kim, Minyong Kim, and Sung Woo Chung. Enhancing energy efficiency

of multimedia applications in heterogeneous mobile multi-core processors. IEEE Transactions on Computers, 66(11):1878–1889, 2017.

[72] Joonho Kong and Kwangho Lee. A DVFS-aware cache bypassing technique for multiple clock domain mobile socs. IEICE Electronics Express, 14(11):20170324–20170324, 2017.

[73] Ludmila I Kuncheva and Lakhmi C Jain. Nearest neighbor classifier: Simultaneous editing and feature selection. Pattern recognition letters, 20(11-13):1149–1156, 1999.

[74] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In 2010 31st IEEE Real-Time Systems Symposium, pages 259–268. IEEE, 2010.

[75] Vuk Lesi, Ilija Jovanov, and Miroslav Pajic. Security-aware scheduling of embedded control tasks. ACM Transactions on Embedded Computing Systems (TECS), 16(5s): 188, 2017.

[76] Huan Li, John Sweeney, Krithi Ramamritham, Roderic Grupen, and Prashant Shenoy. Real-time support for mobile robotics. In The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings., pages 10–18. IEEE, 2003.

[77] Li Li, Jun Wang, Xiaorui Wang, Handong Ye, and Ziang Hu. Sceneman: Bridging mobile apps with system energy manager via scenario notification. In International Symposium on Low Power Electronics and Design (ISLPED), pages 1–6, 2017.

[78] Xueliang Li and John P. Gallagher. An energy-aware programming approach for mobile application development guided by a fine-grained energy model. CoRR, abs/1605.05234, 2016. URL http://arxiv.org/abs/1605.05234.

[79] Wen-Yew Liang and Po-Ting Lai. Design and implementation of a critical speed-based DVFS mechanism for the android operating system. In International Conference on Embedded and Multimedia Computing (EMC), pages 1–6, 2010.

[80] Man Lin, Li Xu, Laurence T Yang, Xiao Qin, Nenggan Zheng, Zhaohui Wu, and Meikang Qiu. Static security optimization for real-time systems. IEEE Transactions on Industrial Informatics, 5(1):22–37, 2009.

[81] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In 25th {USENIX} Security Symposium ({USENIX} Security 16), pages 549–564, 2016.

[82] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM (JACM), 20(1):46–61, 1973.

[83] Cong Liu and J Anderson. Multiprocessor schedulability analysis for self-suspending task systems. Manuscript, The University of North Carolina at Chapel Hill, 2011.

[84] Renju Liu and Mani Srivastava. Protc: Protecting drone's peripherals through arm trustzone. In Proceedings of the 3rd Workshop on Micro Aerial Vehicle Networks, Systems, and Applications, pages 1–6. ACM, 2017.

[85] Yin Liu, Kijin An, and Eli Tilevich. Rt-trust: automated refactoring for trusted execution under real-time constraints. In Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, pages 175–187. ACM, 2018.

[86] Y. Ma, W. Jiang, N. Sang, and X. Zhang. ARCSM: A distributed feedback control mechanism for security-critical real-time system. In Proc. Int. Symp. Parallel and Distributed Processing with Applications, pages 379–386, July 2012.

[87] José Martins, João Alves, Jorge Cabral, Adriano Tavares, and Sandro Pinto. μrtzvisor: A secure and safe real-time hypervisor. Electronics, 6(4):93, 2017.

[88] Alessandra Melani, Maria A Serrano, Marko Bertogna, Isabella Cerutti, Eduardo Quinones, and Giorgio Buttazzo. A static scheduling approach to enable safety-critical openmp applications. In 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), pages 659–665. IEEE, 2017.

[89] Muhammad Hammad Memon, Muhammad Hunain, Asif Khan, Riaz Ahmed Shaikh, and Imran Khan. Power management for android platform by set CPU. In International conference on Computing for Sustainable Global Development (INDIACom), pages 3953–3958, 2016.

[90] Pietro Mercati, Francesco Paterna, Andrea Bartolini, Luca Benini, and Tajana Rosing. WARM: Workload-aware reliability management in linux/android. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 36(9):1557–1570, 2016.

[91] Alexander W Min, Ren Wang, James Tsai, and Tsung-Yuan Charlie Tai. Joint optimization of DVFS and low-power sleep-state selection for mobile platforms. In International conference on Communications (ICC), pages 3541–3546, 2014.

[92] Tanmaya Mishra. Parallelizing trusted execution environments for multicore hard real-time systems. Master's thesis, Virginia Polytechnic Institute and State University, 2019.

[93] Anway Mukherjee and Thidapat Chantem. Energy management of applications with varying resource usage on smartphones. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 37(11):2416–2427, 2018.

[94] Mitra Nasri and Gerhard Fohler. An efficient method for assigning harmonic periods

to hard real-time tasks with period ranges. In 2015 27th Euromicro Conference on Real-Time Systems, pages 149–159. IEEE, 2015.

[95] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. Trustzone explained: Architectural features and use cases. In 2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC), pages 445–451. IEEE, 2016.

[96] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In Cryptographers' track at the RSA conference, pages 1–20. Springer, 2006.

[97] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. IACR Cryptology ePrint Archive, 2002(169), 2002.

[98] M. Pajic, N. Bezzo, J. Weimer, R. Alur, R. Mangharam, N. Michael, G.J. Pappas, O. Sokolsky, P. Tabuada, S. Weirich, and I. Lee. Towards synthesis of platform-aware attack-resilient control systems. In Proc. Int. Conf. High Confidence Networked Systems, pages 75–76, April 2013.

[99] Jurn-Gyu Park, Chen-Ying Hsieh, Nikil Dutt, and Sung-Soo Lim. Quality-aware mobile graphics workload characterization for energy-efficient DVFS design. In Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia), pages 70–79, 2014.

[100] Pratiksha S Patil, Jinalkumar Doshi, and Dayanand Ambawade. Reducing power consumption of smart device by proper management of wakelocks. In International, Advance Computing Conference (IACC), pages 883–887, 2015.

[101] Bo Peng and Nathan Fisher. Parameter adaption for generalized multiframe tasks

and applications to self-suspending tasks. In Embedded and Real-Time Computing Systems and Applications (RTCSA), 2016 IEEE 22nd International Conference on, pages 49–58. IEEE, 2016.

[102] Robert Pettersen, Håvard D Johansen, and Dag Johansen. Secure edge computing with arm trustzone. In IoTBDS, pages 102–109, 2017.

[103] Michael Pinedo. Scheduling, volume 29. Springer, 2012.

[104] Sandro Pinto, Daniel Oliveira, Jorge Pereira, Jorge Cabral, and Adriano Tavares. Freetee: When real-time and security meet. In Emerging Technologies & Factory Automation (ETFA), 2015 IEEE 20th Conference on, pages 1–4. IEEE, 2015.

[105] Sandro Pinto, Tiago Gomes, Jorge Pereira, Jorge Cabral, and Adriano Tavares. Iioteed: An enhanced, trusted execution environment for industrial iot edge devices. IEEE Internet Computing, 21(1):40–47, 2017.

[106] Sandro Pinto, Jorge Pereira, Tiago Gomes, Adriano Tavares, and Jorge Cabral. LTZVisor: TrustZone is the Key. In Marko Bertogna, editor, 29th Euromicro Conference on Real-Time Systems (ECRTS 2017), volume 76 of Leibniz International Proceedings in Informatics (LIPIcs), pages 4:1–4:22, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-037-8. doi: 10.4230/LIPIcs.ECRTS.2017.4. URL http://drops.dagstuhl.de/opus/volltexte/2017/7153.

[107] Alok Prakash, Hussam Amrouch, Muhammad Shafique, Tulika Mitra, and Jörg Henkel. Improving mobile gaming performance through cooperative CPU-GPU thermal management. In Design Automation Conference (DAC), pages 1–6, 2016.

[108] Katerina El Raheb, Christos T Kiranoudis, Panagiotis P Repoussis, and Christos D

Tarantilis. Production scheduling with complex precedence constraints in parallel machines. Computing and Informatics, 24(3):297–319, 2012.

[109] Karthik Rao, Jun Wang, Sudhakar Yalamanchili, Yorai Wardi, and Ye Handong. Application-specific performance-aware energy optimization on android mobile devices. In International Symposium on High Performance Computer Architecture (HPCA), pages 169–180, 2017.

[110] Basireddy Karunakar Reddy, Geoff V Merrett, Bashir M Al-Hashimi, and Amit Kumar Singh. Online concurrent workload classification for multi-core energy management. In 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 621–624. IEEE, 2018.

[111] Basireddy Karunakar Reddy, Amit Kumar Singh, Dwaipayan Biswas, Geoff V Merrett, and Bashir M Al-Hashimi. Inter-cluster thread-to-core mapping and dvfs on heterogeneous multi-cores. IEEE Transactions on Multi-Scale Computing Systems, 4 (3):369–382, 2018.

[112] Concepcio Roig, Ana Ripoll, and Fernando Guirado. A new task graph model for mapping message passing applications. IEEE transactions on Parallel and Distributed Systems, 18(12):1740–1753, 2007.

[113] Masuma Akter Rumi, DM Hasibul Hasan, et al. CPU power consumption reduction in android smartphone. In International conference on Green Energy and Technology (ICGET),, pages 1–6, 2015.

[114] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: what it is, and what it is not. In Trustcom/BigDataSE/ISPA, 2015 IEEE, volume 1, pages 57–64. IEEE, 2015.

[115] Onur Sahin and Ayse K Coskun. Qscale: Thermally-efficient QoS management on heterogeneous mobile platforms. In International Conference on Computer-Aided Design (ICCAD), pages 1–8, 2016.

[116] Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. Real-Time Systems, 49(4):404–435, 2013.

[117] Maria A Serrano, Alessandra Melani, Roberto Vargas, Andrea Marongiu, Marko Bertogna, and Eduardo Quinones. Timing characterization of openmp4 tasking model. In 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), pages 157–166. IEEE, 2015.

[118] Heather Shaw, David A Ellis, Libby-Rae Kendrick, Fenja Ziegler, and Richard Wiseman. Predicting smartphone operating system from personality and individual differences. Cyberpsychology, Behavior, and Social Networking, 19(12):727–732, 2016.

[119] Davesh Shingari, Akhil Arunkumar, and Carole-Jean Wu. Characterization and throttling-based mitigation of memory interference for heterogeneous smartphones. In International Symposium on Workload Characterization (IISWC), pages 22–33, 2015.

[120] Vasileios Spiliopoulos, Akash Bagdia, Andreas Hansson, Peter Aldworth, and Stefanos Kaxiras. Introducing DVFS-management in a full-system simulator. In International Symposium on, Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), pages 535–545, 2013.

[121] Tetsuya Taira, Nobuhide Kamata, and Nobuyuki Yamasaki. Design and implementation of reconfigurable modular humanoid robot architecture. In 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems, pages 3566–3571. IEEE, 2005.

[122] Po-Hsien Tseng, Pi-Cheng Hsiu, Chin-Chiang Pan, and Tei-Wei Kuo. User-centric energy-efficient scheduling on multi-core mobile devices. In Design Automation Conference (DAC), pages 1–6, 2014.

[123] Jeffrey D. Ullman. Np-complete scheduling problems. Journal of Computer and System sciences, 10(3):384–393, 1975.

[124] Roberto Vargas, Eduardo Quinones, and Andrea Marongiu. Openmp and timing predictability: a possible union? In Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, pages 617–620. EDA Consortium, 2015.

[125] Roberto E Vargas, Sara Royuela, Maria A Serrano, Xavi Martorell, and Eduardo Quinones. A lightweight openmp4 run-time for embedded systems. In 2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC), pages 43–49. IEEE, 2016.

[126] Xiaoguang Wang, Yue Chen, Zhi Wang, Yong Qi, and Yajin Zhou. Secpod: a framework for virtualization-based security systems. In Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference, pages 347–360. USENIX Association, 2015.

[127] Yinglong Xia, Viktor K Prasanna, and James Li. Hierarchical scheduling of dag structured computations on manycore processors with dynamic thread grouping. In Workshop on Job Scheduling Strategies for Parallel Processing, pages 154–174. Springer, 2010.

[128] Hui Xu, Yangfan Zhou, Yu Kang, and Michael R Lyu. On secure and usable program obfuscation: A survey. arXiv preprint arXiv:1710.01139, 2017.

[129] Hoeseok Yang and Soonhoi Ha. Power optimization of multimode mobile embedded systems with workload-delay dependency. Mobile Information Systems, 2016, 2016.

[130] Kailiang Ying, Amit Ahlawat, Bilal Alsharifi, Yuexin Jiang, Priyank Thavai, and Wenliang Du. Truz-droid: Integrating trustzone with mobile operating system. In Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '18, pages 14–27, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5720-3. doi: 10.1145/3210240.3210338. URL http://doi.acm.org/10.1145/3210240.3210338.

[131] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. Truspy: Cache side-channel information leakage from the secure world on arm devices. IACR Cryptology ePrint Archive, 2016:980, 2016.