Power-Performance Modeling and Adaptive Management of Heterogeneous Mobile Platforms

by

Ujjwal Gupta

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved April 2018 by the
Graduate Supervisory Committee:

Umit Y. Ogras, Chair
Chaitali Chakrabarti
Michael Kishinevsky
Nikil Dutt

ARIZONA STATE UNIVERSITY

May 2018

ABSTRACT

Nearly 60% of the world population uses a mobile phone, which is typically powered by a system-on-chip (SoC). While the mobile platform capabilities range widely, responsiveness, long battery life and reliability are common design concerns that are crucial to remain competitive. Consequently, state-of-the-art mobile platforms have become highly heterogeneous by combining a powerful SoC with numerous other resources, including display, memory, power management IC, battery and wireless modems. Furthermore, the SoC itself is a heterogeneous resource that integrates many processing elements, such as CPU cores, GPU, video, image, and audio processors. Therefore, CPU cores do not dominate the platform power consumption under many application scenarios.

Competitive performance requires higher operating frequency, and leads to larger power consumption. In turn, power consumption increases the junction and skin temperatures, which have adverse effects on the device reliability and user experience. As a result, allocating the power budget among the major platform resources and temperature control have become fundamental consideration for mobile platforms. Dynamic thermal and power management algorithms address this problem by putting a subset of the processing elements or shared resources to sleep states, or throttling their frequencies. However, an adhoc approach could easily cripple the performance, if it slows down the performance-critical processing element. Furthermore, mobile platforms run a wide range of applications with time varying workload characteristics, unlike early generations, which supported only limited functionality. As a result, there is a need for adaptive power and performance management approaches that consider the platform as a whole, rather than focusing on a subset. Towards this need, our specific contributions include (a) a framework to dynamically select the Pareto-optimal frequency and active cores for the heterogeneous CPUs, such as ARM

big.LITTLE architecture, (b) a dynamic power budgeting approach for allocating optimal power consumption to the CPU and GPU using performance sensitivity models for each PE, (c) an adaptive GPU frame time sensitivity prediction model to aid power management algorithms, and (d) an online learning algorithm that constructs adaptive run-time models for non-stationary workloads.

Dedicated to my family:

Udai Kumar Gupta, Rashmi Gupta,

Dhruv Gupta, and

Ramandeep Kaur

ACKNOWLEDGEMENTS

My deepest gratitude is to my advisor, Dr. Umit Y. Ogras, for the patience, advice and guidance he has offered throughout my study. His insightful recommendations helped me navigate through challenges that I faced during my study. In particular, his tips about software development, writing papers, attending conferences, time management and networking have been extremely useful. This has led to one of the most productive times in my life. It is natural that this dissertation would not be possible without his support.

I am thankful to Dr. Chaitali Chakrabarti, Dr. Michael Kishinevsky, and Dr. Nikil Dutt for taking out time and being in my Ph.D. defense committee.

I am also thankful for encouragement, frequent interactions and help with a number of colleagues and friends at ASU: Dr. Jaehyun Park, Ganapati Bhat, Manoj Babu, Joseph Campbell, Cemil Geyik, Rohit Voleti, Md Moztuba, Hitesh Joshi, Sumit Kumar Mandal, Ugurkan Tursun, Mounica Kothi, Sankalp Jain, Mohit Parihar, Shankhadeep Mukerji, Navyasree Matturu, Spurthi Korrapati, Prasanthi Yelamarthy, Sugam Kapur, Darshan Satyamurthy, Sumit Kumar Mandal, Akhil Arunkumar and Shrikant Singh.

I am thankful to Semiconductor Research Corporation, National Science Foundation and Intel Corporation for funding this research. I enjoyed working closely with Dr. Michael Kishinevsky, Dr. Raid Ayoub, and Dr. Francesco Paterna in the last three years. I also enjoyed working with Dr. Prabhat Mishra and I am indebted for his continued mentorship.

Finally, my parents Udai Kumar Gupta and Rashmi Gupta, brother Dhruv Gupta, and wife Ramandeep Kaur deserve most of the credit for this work. I am grateful for their love, endless support and understanding towards my research career.

TABLE OF CONTENTS

Page

LIST OF TABLES ix

LIST OF FIGURES x

CHAPTER

1 INTRODUCTION 1

1.1 Contributions 2

1.2 Summary of Publications 4

2 DYNAMIC PARETO-OPTIMAL CONFIGURATION SELECTION FOR HETEROGENEOUS MPSOCS 6

2.1 Introduction 6

2.2 Related Research 10

2.3 DyPO Configuration Selection 12

2.3.1 Motivation and Overview 12

2.3.2 Phase-Level Application Instrumentation 16

2.3.3 Data Characterization Methodology 18

2.3.4 Optimal Configuration Classification 20

2.3.5 Online Optimal Configuration Selection 23

2.4 Experimental Results 24

2.4.1 Experimental Setup 24

2.4.2 Classifier Accuracy 27

2.4.3 Runtime Validation of DyPO 28

2.4.4 Improvements in Energy and PPW 33

2.5 Conclusion 37

3 DYNAMIC POWER BUDGETING FOR MOBILE SYSTEMS RUNNING GRAPHICS WORKLOADS 38

CHAPTER Page

3.1 Introduction 38
3.2 Related Research 41
3.3 Power Budget Allocation Mechanism 43
3.3.1 Preliminaries 43
3.3.2 Power Budget Allocation 46
3.3.3 Illustration of the Power Allocation Technique 47
3.3.4 Summary of Overall Operation 50
3.4 Experiment and Simulation Results 50
3.4.1 Hardware Experimental Setup 51
3.4.2 Experimental Results on the Hardware Platform 54
3.4.3 Simulation Framework 58
3.4.4 Simulation Results 61
3.5 Conclusion 63
4 AN ONLINE LEARNING METHODOLOGY FOR PERFORMANCE MODELING OF GRAPHICS PROCESSORS 66
4.1 Introduction 66
4.2 Related Research 69
4.3 Frame Time Characterization 72
4.3.1 Challenges and Notation 72
4.3.2 Frame Time and Counter Data Collection 74
4.4 Frame Time Prediction 78
4.4.1 Differential Frame Time Model 79
4.4.2 Frame Time Sensitivity 81
4.4.3 Offline Feature Selection 84

CHAPTER Page

4.4.4 Online Learning of the Model Parameters 86
4.5 Experimental Results 88
4.5.1 Experimental Setup 88
4.5.2 Offline Feature Selection and ℓ_2 Regularization 90
4.5.3 Online Frame Time Prediction 92
4.5.4 Online Frame Time Sensitivity Prediction 95
4.5.5 Comparison with an Auto Regressive Model using LMS 96
4.5.6 Impact for Dynamic Power Management 97
4.5.7 Overhead Analysis 99
4.6 Conclusion 100
5 STAFF: ONLINE LEARNING WITH STABILIZED ADAPTIVE FORGETTING FACTOR AND FEATURE SELECTION ALGORITHM 109
5.1 Introduction 109
5.2 Related Research 112
5.3 STAFF Online Learning Framework 113
5.3.1 Model Template 113
5.3.2 Stability under Exponential Forgetting 114
5.3.3 Online Feature Selection 116
5.3.4 Adaptive Forgetting Factor 118
5.4 Summary and Complexity Analysis 120
5.5 Experiments 123
5.5.1 Experimental Setup 123
5.5.2 Evaluating the GPU Performance Model 124
5.5.3 Faster Convergence for STAFF 129

CHAPTER Page

5.6 Conclusion 129
6 CONCLUSION 131
REFERENCES 133

LIST OF TABLES

Table Page

2.1 System and application level parameters used in this work. 16

2.2 Data format for each phase. 20

3.1 Notation Table 65

4.1 Summary of the notation used in this chapter 102

5.1 The number of algebraic operations in the SEF and STAFF algorithms in each iteration. 122

5.2 Summary of the baseline algorithms and the proposed STAFF framework. 125

5.3 Three most correlated features used by the STAFF in different time regions of the workload in Figure 5.9. 127

LIST OF FIGURES

Figure Page

2.1 128 different frequency and core configurations of the Blackscholes application showing the trade-off between (a) power consumption and execution time, (b) energy consumption and execution time. 7

2.2 The outline of the proposed approach with an illustrative example. A block of instructions, such as a function call, makes up basic blocks. Our instrumentation groups a sequence of basic blocks into distinct snippets. Finally, each snippet or a sequence of snippets may form workload phases. 14

2.3 PAPI API instrumentation overview. 18

2.4 Training and runtime use of the DyPO classifier. 21

2.5 Implementation of DyPO in Linux Kernel 3.10. 25

2.6 Accuracy of the two classifiers used on the Odroid platform. In multi-threaded benchmarks, -2T and -4T represents two and four threads, respectively. 28

2.7 DyPO-Energy approach compared with the default governors running on the platform. In multi-threaded benchmarks, -2T and -4T represents two and four threads, respectively. 30

2.8 DyPO-Energy, Interactive, Ondemand and Powersave governor comparison for normalized energy consumption. 34

2.9 DyPO-Energy, Interactive, Ondemand and Powersave governor comparison for normalized execution time. 34

2.10 DyPO-Energy, Interactive, Ondemand and Powersave governor comparison for normalized power consumption. 35

Figure Page

2.11 DyPO-Energy, Interactive, Ondemand and Powersave governor comparison for normalized PPW. 35

2.12 Comparison of the normalized PPW obtained using DyPO-Energy approach and Aalsaud et al. [1]. 37

3.1 A sample of the total power consumption and CPU temperature while running the 3D-Mark application. 39

3.2 Example of a CPU-GPU queueing model showing batch buffer and frame buffer. 45

3.3 Summary of the power budgeting technique, showing the steps in each control interval. 51

3.4 Block diagram of the Atom chip [137] used in our experiments. 51

3.5 The sum of the CPU and GPU power consumption for 3D-Mark benchmark showing two levels of power budget. The trace is 15 seconds long, *i.e.*, 300 control intervals. 55

3.6 Deviation from the power budget constraint for different values of the power budget. Each bar reports the average of the error in 3D-Mark, GLbench-Egypt, Citadel, Nenamark2 and Jet-Ski benchmarks. 55

3.7 Experimental results for two different power budget values (50% and 90% of the unconstrained power P_{unconst}). The CPU power fraction of the power budgets is plotted for each of the benchmarks. The GPU power fraction is equal to (1− CPU power fraction). 56

3.8 Comparison of the throughput gain (FPS) achieved with the proposed technique with respect to 1) dynamic heuristic, and 2) static heuristic that allocates 90% of P_{max} to GPU and 10% of P_{max} to CPU. 57

Figure Page

3.9 The average frame rate across all benchmarks for each of the power budget algorithms. 58

3.10 Simulation result of the proposed power budgeting technique showing the sum of the CPU and GPU power consumption and frame rates for different power budget values and workload phases. The GPU is under heavy load to simulate graphics intensive applications like gaming. 59

3.11 Power budget distribution between the CPU and GPU for the three power budget values and workloads. 62

3.12 Static heuristic algorithm result for (a) the sum of the CPU and GPU power consumption and (b) performance for different distributions of GPU/CPU powers. The power budget is set to 2.5 W. Dist=Y/X in the legend indicates that Y% of 2.5 W is allocated to the GPU and X% of 2.5 W to the CPU. . . 64

4.1 The change in frame time for ice-storm application for (a) 200 MHz and (b) 489 MHz GPU frequencies. 67

4.2 (a) Total power consumption of the Intel Minnowboard MAX platform [61] when the GPU is rendering Art3 application at 60 FPS. The crests correspond to the power consumption when the GPU is actively rendering the frames, while the trough correspond to the power consumption when the GPU is in sleep state. (b) Zoomed portion, which shows three frames in the first 50ms. The width of the peaks give the time the GPU is actively computing the frame. (c) Frame time distribution for kernel and power instrumentations for Art3 application. 73

4.3 The frame time distribution obtained for rendering the same frame and rendering multiple similar frames. 76

Figure Page

4.4 The proposed methodology for collecting a rich set of training and test data. Each frame is repeated n_r times for every configuration. 77
4.5 Frame time and hardware counter values for the RenderingTest application with increasing GPU frequency at four different frame complexities. 101
4.6 Frame time for the RenderingTest application with increasing frame complexity at four different GPU frequencies. 101
4.7 Adaptive filtering approach showing the update in parameters a_i based on error between the actual change in frame time and prediction. 103
4.8 Cross-validated LASSO regression result for; (a) the change in mean squared error of the frame time prediction with increasing η values, and (b) the change in the number of selected features with increasing η values. 103
4.9 Correlation between the selected features and the difference in the frame time $t_k - t_{k-1}$. 104
4.10 Frame time prediction error for RenderingTest and Art3 applications for different values of the ℓ_2 regularization parameter μ. The black markers show the mean value of the error and the whiskers show the one standard deviation boundaries. 104
4.11 Frame time prediction for the RenderingTest app. 104
4.12 Frame time prediction for the 3DMark Ice Storm application running at (a) 200 MHz, (b) 489 MHz. 105
4.13 Frame time prediction for the BrainItOn application running at 200 MHz. 105

4.14 Mean absolute percentage errors in the frame time for the Android applications using the three algorithms: RLS, RLS+Offline, and DCD-RLS. . . . 106
4.15 Comparison of mean absolute percentage error in frame time for all Android applications combined. . . . 106
4.16 Frame time prediction while running YouTube and Chain reaction game running simultaneously on Moto-X smartphone. . . . 106
4.17 Predicted and actual frame times for RenderingTest application when f_{new} is one level higher. . . . 107
4.18 Frame time prediction error in RenderingTest application for multiple frequency jumps. . . . 107
4.19 Sensitivity of frame time with respect to frequency for (a) RenderingTest and (b) Art3 applications. . . . 107
4.20 The proposed RLS technique converges in only $50ms$ compared to the AR-LMS technique that converges in $1.6s$ for the Icestorm application. 108
4.21 Normalized energy consumption of the Ondemand governor and our RLS-based policy normalized to the Oracle-based policy. . . . 108
4.22 Overhead time as a function of the number of features for the RLS and DCD-RLS algorithm. . . . 108
5.1 A non-stationary GPU workload (a) example and (b) analysis using autocorrelation function (ACF). . . . 110
5.2 The y-axis is in $\log_{10}$scale. ℓ_1 norm of the inverse of the correlation matrix $\mathbf{R}$ shows unstable behavior for $\alpha = 0$ and stability for $\alpha = 10^{-5}$, respectively. . . . 115

Figure Page

5.3 Online STAFF framework has superior tracking performance to offline feature selection. 118

5.4 Illustration of the entropy-based change detection. The solid-line shows the entropy, while the and dashed-lines show the likelihoods of feature 1 and feature 2, respectively. 120

5.5 The STAFF framework adapts much faster to the new workload compared to $\lambda = 0.99$. In addition, it does not possess the local erroneous tracking of a_0 caused by $\lambda = 0.9$. 120

5.6 Summary of the STAFF algorithm. 121

5.7 STAFF framework has 3.2× lower error (right axis) in frequency sensitivity, and 6.5× lower complexity compared to the SEF algorithm for $M = 17$. The errors are computed using the non-stationary workload employed in Section 5.5.2 122

5.8 Comparison of the STAFF framework against *constant* forgetting factor approaches. 124

5.9 Comparison of the STAFF framework against *adaptive* forgetting factor approaches. 124

5.10 Normalized RMS error in the frequency sensitivity estimates of the algorithms in different workload regions. Errors are normalized with respect to the best baseline approach (SEF+Offline_FS*). 127

5.11 Analysis of correlation coefficients, likelihoods, and entropy of the hardware counter features for STAFF algorithm at 2 minutes time of Figure 5.9. (a) The correlation coefficients become equal in the time interval immediately after a workload change occurs at time 2 minutes. (b) The likelihood values become equal in the same time interval. (c) The entropy for the set of the hardware counter features also changes and peaks in the same time interval. 128

5.12 Comparison of the STAFF and Fast-STAFF algorithm. 129

Chapter 1

INTRODUCTION

More than half of the world's population uses mobile systems for a variety of tasks, such as calling, video conferencing, navigating, and gaming [133]. The users of mobile systems primarily care about the responsiveness, long battery life and reliability of the mobile platforms. Runtime management of these concerns is necessary, because the set of active applications and their requirements change dynamically. Dynamic thermal and power management (DTPM) techniques manage the trade-off between performance, power consumption and temperature to provide a desirable user experience on a mobile system. Among the many components of a mobile system, the System-on-a-chip (SoC) is one of the most power hungry and hot components. As a result, dynamically managing the power consumption of a SoC is crucial to deliver competitive performance. Furthermore, the SoC itself is a heterogeneous resource that integrates many processing elements (PE), such as CPU cores, GPU, video, image, and audio processors [42, 43]. Therefore, it becomes important to manage power consumption of these PEs, since the CPU cores do not dominate the platform power consumption under many application scenarios [14].

DTPM techniques for mobile phones have received unprecedented attention in the last decade [121]. This is evident from the industry wide trend to provide more number of OS-level hardware configuration knobs, such as frequency and number of cores for the CPU and GPU [79, 86, 110]. Furthermore, these configurations are expected to increase in the future, to provide power management architects more flexibility in managing the power consumption of even more PEs. Therefore, techniques that consider power management of more than one PE are now critical [57, 70].

Our overarching goal is to design theoretically grounded and practical approaches that will comprehend the whole platform and adapt to workload and temperature variations. A strong theoretical foundation is necessary to provide stability, power and performance guarantees, while maintaining scalability in terms of platform components, such as the number of cores. To be effective, new approaches should have low overhead and be amenable to integration with existing hardware, firmware and software stacks. Towards these goals, we develop novel techniques for modeling, analysis, and optimization of power consumption, energy, performance per watt, and performance.

1.1 Contributions

Our first contribution is a dynamic power management technique for a recently introduced single ISA big.LITTLE heterogeneous CPU system [51]. Power management of heterogeneous systems involve managing a number of configurations, i.e., the number of big and little cores, and their frequencies. Dynamically selecting the optimal configuration is a challenging task because the configurations change as the composition of the active applications and their phases vary. Moreover, finding the optimal configuration as a function of workload is difficult (even offline), since it requires running precisely the same workload at each possible configuration, especially for CPUs [151]. Therefore, we develop a framework to select the Pareto-optimal configurations at runtime, using multinomial logistic regression classifiers that are built offline [46]. Experimental evaluation of our technique shows substantial gains in performance per watt compared to the default and state-of-the-art techniques [1, 106].

While the CPU is one of the most important components of a SoC, a number of mobile applications, such as games critically depend on the GPU for rendering. Therefore, in our second contribution, we focus on power management of the CPU

and GPU together for graphics workloads. High graphics performance comes at the cost of higher power consumption, which elevates the temperature of the mobile system due to limited cooling solutions. To avoid thermal violations, the system needs to operate within a power budget. Since the power budget is a shared resource, there is a strong demand for effective dynamic power budgeting techniques. This chapter presents a novel technique to efficiently distribute the power budget among the CPU and GPU cores, while maximizing performance. The proposed technique is evaluated using an Intel Baytrail platform [62] running industrial benchmarks, and an in-house simulator [47].

Finally, integrated GPUs have become an indispensable component of mobile processors due to the increasing popularity of graphics applications. The GPU frequency is a key factor both in application throughput and mobile processor power consumption under graphics workloads. Therefore, dynamic power management algorithms have to assess the performance sensitivity to the GPU frequency accurately. Since the impact of the GPU frequency on performance varies rapidly over time, there is a need for online performance models that can adapt to varying workloads. To address this need, a number of performance models have been proposed [27, 28, 70, 111]. Yet, these models do not generalize well to a larger set of workloads due to offline training and coarse-grain inputs, such as utilization. In stark contrast, we construct a light-weight adaptive runtime performance model that predicts the frame processing time of graphics workloads at runtime without apriori characterization. We employ this model to estimate the frame time sensitivity to the GPU frequency, *i.e.*, the partial derivative of the frame time with respect to the GPU frequency. The proposed model does not rely on any parameter learned offline. We also experimentally validate the model on an Intel Minnowboard MAX platform [61] running common GPU benchmarks [40].

This dissertation summarizes our contributions that aid the power management of heterogeneous systems. More precisely, our specific contributions are as follows:

- A framework to dynamically select the Pareto-optimal frequency and active cores for the heterogeneous CPUs, such as ARM big.LITTLE architecture [46, 51],
- A dynamic power budgeting approach for allocating optimal power consumption to the CPU and GPU using performance sensitivity models for each PE [47],
- An adaptive GPU frame time sensitivity prediction model to aid power management algorithms [40, 109].
- A novel online learning framework for recursive parameter estimation [39].

The rest of the dissertation is organized as follows. Chapter 2 presents the dynamic Pareto-optimal configuration selection framework for heterogeneous MpSoCs. Chapter 3 details the dynamic power budgeting approach for CPU and GPU running graphics applications. Chapter 4 presents the technique for online learning of GPU frame time model. Chapter 5 presents a new online learning algorithm that can perform online feature selection and adaptive forgetting with stability. Finally, Section 6 concludes the dissertation prospectus with discussion on future work.

1.2 Summary of Publications

This dissertation is a collection of the research manuscripts written by the author in the area of dynamic power management for heterogeneous computing platforms [39, 40, 46, 47, 109]. Besides working in this area, the author also extensively worked in the area of flexible hybrid electronics. Flexible hybrid electronics includes the development of mechanically flexible, printed and stretchable electronics [44].

While rapid advancement is well underway at the device and circuit levels, researchers have yet to envision the system design in a flexible form. We introduce the concept of *Systems-on-Polymer (SoP)* based on flexible hybrid electronics (FHE) to combine the advantages of flexible electronics and traditional silicon technology [41, 45]. First, we formally define flexibility as a new design metric in addition to existing power, performance, and area metrics. Then, we present a novel optimization approach to place rigid components onto a flexible substrate while minimizing the loss in flexibility. We show that the optimal placement leads to as much as 5.7× enhancement in flexibility compared to a naïve placement. We confirm the accuracy of our models and optimization framework using a finite element method (FEM) simulator. Finally, we demonstrate the SoP concept using a concrete hardware prototype, and discuss the major challenges in the architecture, design of SoPs, and applications [10, 95].

Chapter 2

DYNAMIC PARETO-OPTIMAL CONFIGURATION SELECTION FOR HETEROGENEOUS MPSOCS

2.1 Introduction

State-of-the-art smartphones and tablets have to satisfy the performance requirements of a diverse range of applications under tight power and thermal budget [19, 128]. The number of power management configurations offered by MpSoCs, such as the number of voltage-frequency levels and active cores, have been increasing steadily to adapt to these dynamically varying requirements. For example, octa-core big.LITTLE architectures have 20 different CPU core configurations that can be selected at runtime. Combined with the voltage and frequency levels, this leads to more than 4000 dynamic configurations to consider during optimization. This huge collection results in more than one order of magnitude variation in both power consumption and performance, as shown in Figure 2.1(a). Moreover, the definition of the optimality can change depending on the context. For instance, users prefer to maximize the responsiveness (i.e., performance) for interactive applications, while minimizing the energy becomes the priority when the platform is running out of power. Therefore, it is crucial to identify the optimal configuration at runtime.

Dynamically selecting the optimal configuration is a challenging task aggravated by two major factors. First, the design space is large for a runtime evaluation and exploration. Therefore, an exhaustive search is prohibitive due to significant overhead associated with exploration. Second, and more importantly, the optimal choice is a strong function of the workload, which itself varies dynamically [13]. For example,

bringing the data from memory faster is important upon launching the application, but processing time starts dominating later on. Similarly, the application may go through CPU- and memory-bound phases during its lifetime. Consequently, the optimal configuration changes as the composition of the active applications and their phases vary.

Chip designers and power management architects spend significant effort to attain the optimal power-performance trade-off. For example, Figure 2.1(a) plots power consumption and execution time of a multi-threaded application for 128 different core and operating frequency configurations. We clearly see that many configurations are close to the Pareto-optimal curve. Frequency governors integrated in the OS-stack leverage this fact effectively to deliver the desired trade-off. For instance, the interactive and on-demand governors increase the frequency whenever core utilizations exceed a threshold to maximize the performance, while the powersave governor chooses the minimum operating frequency to minimize power consumption [106]. Similarly, the dynamic power management algorithms, such as cpuidle, increase (decrease) the number of active cores when the core utilizations are above (below) tunable thresh-

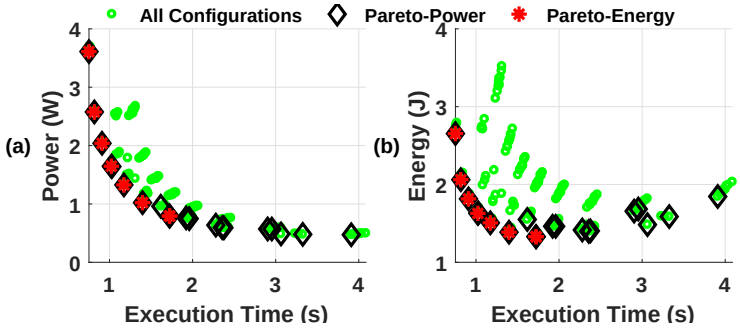

Figure 2.1: 128 different frequency and core configurations of the Blackscholes application showing the trade-off between (a) power consumption and execution time, (b) energy consumption and execution time.

olds [8, 105]. Hence, these highly optimized governors can dynamically scale the number of active cores and frequency to optimize the power-performance trade-off. However, none of these approaches can guarantee optimality with respect to other metrics, such as energy consumption. For instance, Figure 2.1(b) shows that many Pareto-optimal configurations in the power-performance plane are far away from the *Pareto curve in the energy-performance plane*. Moreover, a governor that chooses the lowest power configuration results in 39% more energy consumption and 126% slower execution with respect to the minimum energy configuration. Our experimental results reveal similar trends for default governors for many other metrics, such as performance per watt and instructions per second. Therefore, there is a strong need for runtime algorithms that can choose the optimal configuration with respect to a given metric as a function of the workload.

This chapter presents a comprehensive methodology to choose optimal core and frequency configuration at runtime as a function of workload characteristics. Existing approaches rely on core utilizations to make decisions in single steps [106]. In strong contrast, we employ a classifier that chooses the optimal configuration for a given workload phase characterized with a diverse set of performance counters available on the target platform.

Our major contributions towards enabling and validating the proposed methodology are as follows:

- Instrumentation (Section 2.3.2): Finding the optimal configuration as a function of workload is difficult (even offline), since it requires running *precisely the same* workload at each possible configuration. One could run a given application at each possible configuration and collect statistics at uniform time intervals. However, the workload in each time interval would be different for each configuration, since the instructions are processed at different speeds. Therefore, the first step of the proposed

methodology is instrumenting the applications using the LLVM [78] compiler infrastructure and PAPI calls [92]. This instrumentation, which has less than 1% overhead, enables us to collect a vast amount of characterization data for each workload snippet [1] .

- Characterization (Section 2.3.3 & 2.3.4): The second step is to collect characterization data using the instrumented applications. In this work, we collected power consumption, processing time and six performance counters for a total of 4,467 workload snippet using 18 different applications. In the third step, we use the power consumption and processing time information to identify the optimal configuration for each of the 4,467 workload snippet with respect to any metric, such as energy, which can be expressed in terms of this information. Finally, the characterization data is used to find classifiers that map each workload snippet to its optimal configuration.

- Runtime selection (Section 2.3.5): Our final step is to develop a new governor that implements the classifier for each metric of interest. The user can easily choose any of the classifiers in this unified governor at runtime by setting a variable at user space. The same features (i.e., performance counters and core utilizations) used for characterization are collected at runtime. Then, the features are fed to the classifier to find the optimal configuration.

- Experimental validation (Section 2.4): We present an extensive set of evaluations using 18 single- and multi-threaded applications running on Odroid XU3. We obtain on average 49%, 45% and 6% lower energy consumption compared to the interactive, ondemand, and powersave governors, respectively. Our approach also outperforms the powersave governor by achieving lower execution time, but has longer execution time than interactive and ondemand governors, as explained in Section 2.4.

[1] In this chapter, a workload snippet is a sequence of basic blocks with sizes varying from 5k to 100M instructions, as explained in Section 2.3.2. A group of consecutive snippets make up a workload phase. Each snippet is similar to a micro-benchmark.

The rest of the chapter is organized as follows: Section 2.2 presents the related work. Section 2.3 lays out the groundwork required for collecting meaningful experimental data and the framework for the proposed technique for optimization. Section 2.4 discusses the experimental results. and Section 2.5 presents the conclusion.

2.2 Related Research

Widespread use of mobile platforms in the last decade is enabled by advanced power management techniques, including dynamic core and uncore scaling [16, 73, 100], cache reconfiguration, task partitioning, task scheduling, and power budgeting [47, 53, 141, 144]. Significant number of these power management techniques focus on power and performance optimization through dynamic power management (DPM) and dynamic voltage, frequency scaling (DVFS). DPM consists of a set of algorithms that selectively turns off system components that are idle, such as controlling the number of active cores in the system depending on their utilization [8]. Similarly, DVFS-based schemes control the operating frequency of a core based on the utilization [54, 94, 106]. For example, millions of commercial mobile platforms run the ondemand and interactive governors [106]. However, these techniques do not guarantee optimality with respect to a given metric such as energy consumption. These approaches typically perturb the configuration by a single predetermined step. For example, interactive and ondemand governors increase (decrease) the frequency of the processor if the utilization is above (below) a certain threshold [106]. The work presented in [146] proposes a technique to maximize the performance within a given power budget by estimating Pareto-optimal solutions dynamically. This approach relies on analytical power consumption and instructions per second model to find the Pareto-optimal frequency configurations of homogenous architectures. In contrast, our approach finds the Pareto-optimal core and frequency configuration in heteroge-

neous architectures using an extensive set of hardware measurements and multinomial logistic regression. Hence, our approach combines DVFS and DPM by setting the operating frequency/voltage and the type and number of active cores simultaneously.

Recently, a number of studies have focused on workload-aware DPM and DVFS together [1, 17, 23, 26, 29, 80, 152]. These techniques choose the best or a mixture of the two strategies to optimize the mobile platform. For instance, the technique proposed in [1] first derives the power and performance models using multivariate linear regression for each different frequency and application. Then, these models are used to determine an optimal performance per watt configuration for an application at runtime. Similarly, the work in [26] proposes an online learning method to select the best-performing DPM policy together with DVFS settings called experts, for a single CPU core. At runtime, the controller characterizes the workload based on energy and cycles-per-instruction models to choose the best-performing expert. The work in [23] proposes a new Linux scheduler to optimize the power consumption under a throughput constraint. Their approach is specifically designed for parallel applications with computation intensive loops. Similarly, the approach proposed in [152] focuses on a group of applications related to web browsing for heterogeneous platforms. They build linear regression models for performance and energy consumption, and then use them to schedule webpages for minimizing the energy consumption of the system. Several recent techniques have also considered applying classification-based methods for the frequency and core selection. For example, the work in [17] proposes a technique for homogeneous server systems, which uses logistic regression to find thread packing and frequency such that the system remains within a power budget. Similarly, the work in [29] uses binning-based classification for identifying the degree of memory- and compute-boundedness of the tasks. Then, these tasks are allocated based on the predicted power and performance to the CPU cores for

minimizing the power consumption under a throughput constraint. However, none of the above methods use phase-level instrumentation, which is necessary to identify the optimal configurations for a given workload.

Phase-level performance and power analysis provide a fine grained and reliable information about the workload, as we describe in Section 2.3.2. This information enables accurate power and performance models across different platforms [151] and practical power management algorithms [65]. For example, using the phase-level analysis one can collect statistics on one platform and use it to predict the power and performance on another platform [151]. This leads to significant improvements in the accuracy of the models by using this insight compared to an approach that uses aggregate application statistics. Therefore, in contrast to the other DVFS and DPM approaches, our work leverages the use of phase-level offline characterization for a number of benchmarks to find the Pareto-optimal configurations for each phase. Then, we build classifiers that map the characterized feature data to the Pareto-optimal configurations. Finally, the classifier is used at runtime to select the optimal configuration for a new application phase. In our experimental evaluations, we observe substantial numerical gains in performance per watt compared to a recently proposed algorithm [1] and the default governors.

2.3 DyPO Configuration Selection

2.3.1 Motivation and Overview

Modern MpSoCs offer a staggering number of configuration knobs. For example, the recently introduced Samsung Exynos 5422 MpSoC based on ARM big.LITTLE architecture offers four little (A7) and four big (A15) cores that can operate at 13 and 19 different frequencies, respectively [51]. Furthermore, the voltage of each of

the core clusters scales with frequency. Since at least one little core has to remain active at all times, this leads to a total of (4×13×4×19) + (4×13) = 4004 different frequency and core configurations. Different configurations lead to a huge variation in power consumption and performance, as shown in Figure 2.1. Moreover, any given application workload consists of multiple workload phases [126]. For example, lower CPU frequencies may save power during a memory-intensive phase. In contrast, CPU-intensive phases with many active threads are likely to benefit more from higher frequencies and number of cores. Therefore, different configurations may become optimal with respect to a given metric as the workload varies at runtime [13].

We denote the set of all possible configurations by $\mathbf{C}$, and the configuration at time k with $c_k \in \mathbf{C}$. Each feasible configuration can be represented by $c_k = \{n_{L,k}, f_{L,k}, n_{B,k}, f_{B,k}\}$, where the elements represent the number of active little cores, the frequency of little cores, the number of active big cores, and the frequency of big cores, respectively. Similarly, we denote the set of phases encountered during the lifetime of an application by $\mathbf{P}$, and the phase at time k with $p_k \in \mathbf{P}$. Our optimization goal can be expressed as:

$$\begin{aligned} \textbf{Find} \quad & f : \mathbf{P} \ni p_k \mapsto c_k^* \in \mathbf{C} \\ & \textit{where } c_k^* \in \mathbf{C} \text{ is the optimal configuration} \\ & \textit{for workload phase } p_k \in \mathbf{P} \end{aligned} \tag{2.1}$$

Identifying the optimal configuration c_k^* at runtime for each phase p_k is a daunting task due to the large number of workloads and configurations. For example, the Basicmath application has three phases, and identifying the optimal configuration of each phase would mean searching through 4004^3 ($\approx 6 \times 10^{10}$) different possibilities for the entire application. Clearly, searching through this combinatorial space is intractable at runtime. Furthermore, the definition of the optimality may change

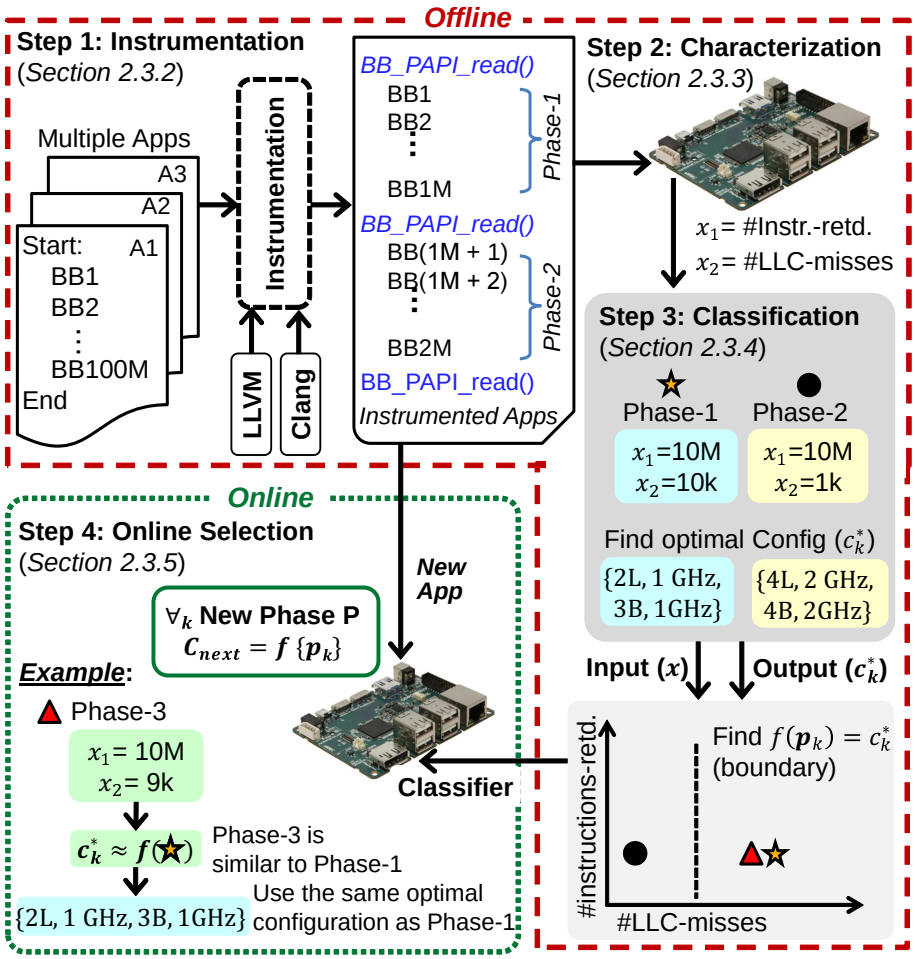

Figure 2.2: The outline of the proposed approach with an illustrative example. A block of instructions, such as a function call, makes up basic blocks. Our instrumentation groups a sequence of basic blocks into distinct snippets. Finally, each snippet or a sequence of snippets may form workload phases.

over time depending on the application scenario. For example, minimizing the energy consumption becomes a priority when the battery is running low. Hence, there is a strong need to dynamically identify the optimal configuration c_k^* for a given optimization objective at any point in time.

Overview and illustrative example: We start with an overview and illustrative

example, before detailing the proposed approach. First, we instrument the target application to divide the workload into groups of basic blocks called snippets. This step enables us to collect power and performance statistics of each snippet at runtime, as illustrated in Figure 2.2. For example, consider an application code with 100 million basic blocks (BB1 to BB100M) where each basic block is a sequence of instructions. The instrumentation in this example inserts special BB_PAPI_read() basic blocks that call the PAPI APIs for reading hardware counters and system statistics every 1 million basic blocks. A pair of BB_PAPI_read() basic blocks create a boundary for different snippets of an application. Each snippet or a sequence of snippets may form distinct phases. Offline instrumentation is followed by the characterization step, where we collect extensive power consumption and performance data for a large variety of single- and multi-threaded applications (Section 2.3.3). More specifically, we collect the data listed in Table 2.1 while repeatedly running each application using different configurations supported by the platform. Then, this data is used to identify the optimal configuration for each workload snippet. The third step is to design a classifier using this characterization data (Section 2.3.4). For example, consider two different snippets, the first with 10K *LLC-misses* (high) and the second with 1K *LLC-misses* (low). Suppose that the characterization step reveals the optimal configurations as {2L, 1 GHz, 3B, 1 GHz } and {4L, 2 GHz, 4B, 2 GHz }, respectively. The classification step uses these data points to design a classifier $f : \mathbf{P} \ni p_k \mapsto c_k^* \in \mathbf{C}$ that maps different snippets to the optimal configurations at runtime. The plot in the lower right corner of Figure 2.2 illustrates a potential classifier that can clearly separate these two snippets. The final step is using the classifier online to determine the optimal configuration for any workload encountered at runtime (Section 2.3.5). As an example, assume that the system encounters Phase-3, which has 9K *LLC-misses* and similar number of *instruction-retired* with Phase 1 and Phase 2. Since Phase-3

Table 2.1: System and application level parameters used in this work.

Application Level Parameters	**System Level Parameters**
Instructions Retired	Per Core CPU Frequency
CPU Cycles	Per Core CPU Utilization
Branch Miss Prediction	little, big, GPU and DRAM Power Consumption
Level 2 Cache Misses	Number of Active Cores
Data Memory Access	Execution Time
Noncache External Memory Request	

is closer to Phase-1 characterized offline, the classifier will assign it the same optimal configuration of {2L, 1 GHz, 3B, 1 GHz }. While our illustrative example is simple, the real problem is multidimensional and far more challenging than creating simple visual boundaries between phases. The rest of this section detail these four steps employed in the proposed methodology.

2.3.2 Phase-Level Application Instrumentation

Platform designers provide a rich set of hardware and software counters that can be accessed at runtime to identify different workload phases. The PAPI infrastructure provides user level APIs that can be inserted within the application to capture these counters at runtime [92]. In addition to the performance counter information provided by PAPI, it is also important to capture system behavior during the same interval. Therefore, we also log important features, such as the total CPU power consumption, core frequencies, core utilizations, and execution time, by modifying the PAPI API. The system and application level parameters employed in this work are listed in Table 2.1.

To accurately instrument applications with PAPI APIs, we use the LLVM compiler infrastructure, which has the functionality to analyze any given source code at different granularities, such as module level, function level, and basic block level [78]. LLVM treats any input source as a single block of module that can be broken down into functions. Each of these functions contains different basic blocks that subsequently contain assembly instructions. Instrumenting at the function level is too coarse, while instrumentation at the instruction level is too fine-grained. Therefore, we utilize LLVM with *clang* compiler [77] to analyze and instrument PAPI calls at critical basic blocks within an application to collect the hardware counters at runtime.

Figure 2.3 illustrates the process of instrumenting any benchmark with PAPI calls using LLVM and *clang* compiler. The first step is an instrumentation pass source file in LLVM that can identify existing functions and basic blocks, and add new functions (PAPI APIs) for any application. Then, we use Cmake/Make utilities to compile the LLVM instrumentation pass to get a custom library object file. Finally, we use the *clang* compiler to compile the benchmark with the custom library as an additional input. This generates an output object file that has PAPI APIs instrumented at different basic blocks. Note that our instrumentation process is independent of how the application code is written, as it relies specifically on analyzing the basic blocks, which are the building blocks of any application and a widely used syntax analysis terminology in the compiler domain.

Instrumenting single-threaded workloads requires identifying the critical basic blocks and then adding simple PAPI calls. While instrumenting the multi-threaded benchmarks, we tie each thread to its own performance counter values. We achieve this with the help of PAPI APIs, which provide specific calls to register threads that can maintain their own counter data. Since multi-threaded workloads also have phases that only have single threads, we ensure that our instrumentation can cap-

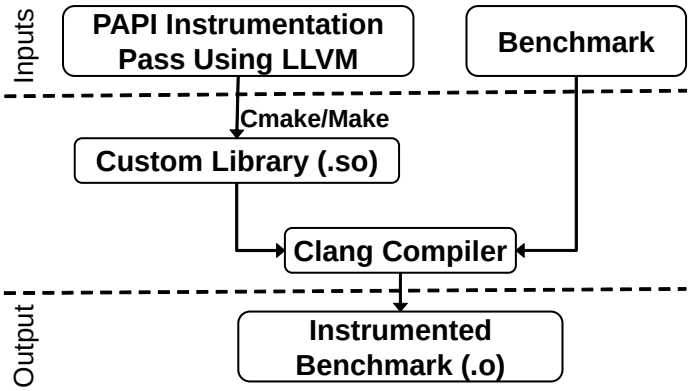

Figure 2.3: PAPI API instrumentation overview.

ture such phases as well. At the time of logging the hardware counter values along with system performance, we also capture the thread IDs and time-stamp of data collection. This methodology ensures that we are able to analyze both single- and multi-threaded phases of any workload. In practice, inserting PAPI APIs are expected to introduce extra instructions as overhead. Therefore, we ensure that the overhead introduced with these API calls is negligible, as detailed in Section 2.4.1. Overall, the process of instrumentation enables us to capture the critical regions that provide useful information regarding different phases of an application running on any platform.

2.3.3 Data Characterization Methodology

Once the benchmarks are instrumented with the PAPI APIs, we collect data for different frequency and core configurations. We first set the highest frequency and core configuration, i.e., 2 GHz for the big cores with all eight cores active. Then, we run three iterations of each benchmark at this frequency and core configuration. Next, we step down the frequency of the big core cluster while maintaining the number of active cores. We repeat this process for each benchmark included in the study. After

this, we reduce the frequency level by one, and repeat this process for all supported frequency levels and core configurations. Since the number of total configurations is large even for offline analysis, we use a representative data set obtained by running each benchmark three times with 4×4×8=128 different core and frequency configurations [2] . This selection includes all core configurations (4×4) from 1L+1B to 4L+4B. We include at least one little and one big core, since we are interested in maintaining the heterogeneity of the system. We sweep the frequency uniformly from 0.6 GHz to 2 GHz in steps of 0.2 GHz for all 16 core configurations. Frequencies lower than 600 MHz are not included, since they are rarely energy optimal. Indeed, default Android governors also do not utilize lower frequencies. That is, the lowest power configuration in our experimental setup is {1L, 0.6 GHz, 1B, 0.6 GHz} and the highest performance configuration is {4L, 1.4 GHz, 4B, 2 GHz}. We run the entire application from start to end for all the selected configurations. Therefore, all the relevant phases are considered irrespective of the application. In this work, our specific knowledge about the target platform is used to choose the frequency configurations. In general, one can also apply formal approaches to select a representative set of configurations [103, 104]. On profiling three iterations of 18 benchmarks for 128 different configurations lead to a total of 6,912 different benchmark runs. We always re-boot the system before starting the data collection process for each benchmark to ensure consistency of the platform environment. Finally, we collect the characterization data for each workload snippet following the format shown in Table 2.2.

[2] Time spent for collecting data for 128 configurations on Odroid XU3 is typically about 1-2 hours per benchmark.

Table 2.2: Data format for each phase.

Time-stamp	Power Consumption	# Active Cores	CPU Frequency	Perf. Cntr 1	...	Perf. Cntr N	Core Utilizations

One row for each workload snippet, frequency, little core and big core configuration
Total number of rows per phase of a benchmark= $n_{\text{big}} \times n_{\text{little}} \times n_{\text{freq}} \times n_{\text{repeat}}$

2.3.4 Optimal Configuration Classification

After the characterization is complete, we first find the Pareto-optimal configurations for each characterized workload snipped with respect to a given optimization goal. Then, this data combination, i.e., (snippet, optimal configuration) is used to design a classifier. Finally, this classifier is stored on the platform and used at runtime to select the optimal configuration, as detailed in Section 2.3.5.

Optimization Goal

Energy consumption and responsiveness are of primary importance in mobile systems [141]. Furthermore, optimizing them also improves performance per watt (PPW). Therefore, we consider a bi-objective optimization problem of minimizing the energy consumption $E(c_k, p_k)$ and execution time $t_{\text{exe}}(c_k, p_k)$ for program snippet p_k and configuration c_k. The optimal cost $J(p_k)$ for this bi-objective problem can be written as follows:

$$J(p_k) = \min_{c_k} \left[E(c_k, p_k) + \mu t_{\text{exe}}(c_k, p_k) \right] \tag{2.2}$$

where $\mu \geq 0$ is a weight between the energy and execution time that determines the relative importance of the two objectives. For example, when μ is small, the optimization problem essentially turns into minimization of energy (DyPO-Energy), and when μ is large, the optimization problem minimizes the execution time (DyPO-Performance). Any μ value in between will lead to minimizing the energy consumption

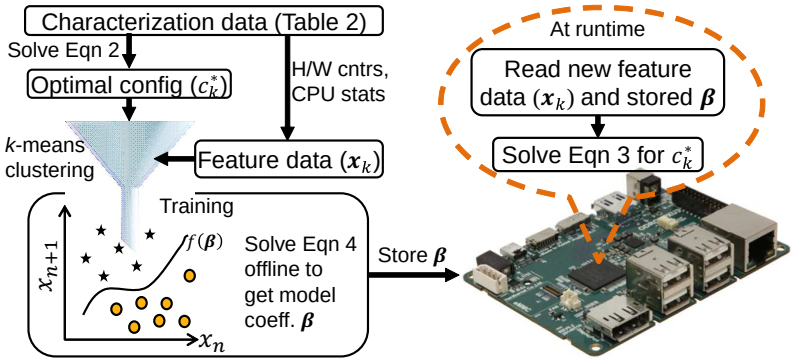

Figure 2.4: Training and runtime use of the DyPO classifier.

with some other execution time constraint. *More importantly*, our classification does not depend on the structure of Equation 2.2, as described next. Therefore, we can compute the Pareto-optimal configurations c_k^* for each snippet p_k for an arbitrary optimization objective that combines energy, execution time, instructions per cycle and power consumption.

Design of the Classifier

Once the Pareto-optimal configuration c_k^* for each snippet p_k is identified using Equation 2.2, the next task is to map different snippets to their optimal configurations using the function $f : \mathbf{P} \ni p_k \mapsto c_k^* \in \mathbf{C}$. We utilize multinomial logistic regression classification technique for this purpose due to its simple implementation in the kernel. However, any other supervised machine learning classification technique can be used to the same effect.

To train the logistic regression classifier, we need to use input features and associated output labels, as shown on the upper left corner of Figure 2.4. The inputs to the classifier are five hardware counters, shown in Table 2.1 normalized with *instructions-retired*, the sum of the utilizations of the little cores, sorted utilizations of the big cores, and one bias term. The output labels are the optimal configurations found

with respect to the criterion in Equation 2.2. Note that two different snippets can map to the same optimal configuration. Hence, an approach that arbitrarily assigns a supervisory response (optimal configuration) to the features would fail to create a good mapping function *f*. To avoid this, we first employ k-means clustering to find natural clustering in the data set [34]. Then, we assign the most frequently occurring optimal configuration in each of the clusters as their output labels. This can also be performed hierarchically with multiple levels of k-means clustering and classification. For example, we use two highly accurate classifiers with three classes each in our experiments, as explained in Section 2.4.2. After the input features and output labels are determined, we design the classifier, as described next.

The conditional probability of the occurrence of a Pareto-optimal configuration $c_k^* \in \mathbf{C}$ given an input $\mathbf{x}_k = [x_1, x_2, x_3, ..., x_N]$, can be represented as $\Pr(\mathbf{C} = c_k^* | \mathbf{x} = \mathbf{x}_k)$. We express the conditional probability for each Pareto-optimal configuration using a logistic function as follows [34]:

$$\Pr(\mathbf{C} = c_k^* | \mathbf{x} = x_k) = \frac{e^{\beta \mathbf{x_k}}}{1 + e^{\beta \mathbf{x_k}}} \tag{2.3}$$

where $\beta = [\beta_0, \beta_1, ..., \beta_N]$ are the regression coefficients *learned offline* using the characterized data for each workload snippet. The regression coefficients are estimated by maximum likelihood, using the known conditional likelihoods for a class $\mathbf{C}$ given features $\mathbf{x}$ (training data). When the total number of data points (i.e., number of workload snippets $\times$ number of configurations) is M, the likelihood function can be written as:

$$\ell(\beta) = \prod_{k=1}^{M} \Pr(\mathbf{C} = c_k^* | \mathbf{X} = x_k) \tag{2.4}$$

Since the maximum likelihood function in Equation 2.4 is non-linear, we use the mnrfit function in Matlab to solve for the β values offline. Then, we store the β values as look-up tables in the platform, and use them for selecting the optimal configurations

at runtime, as illustrated in Figure 2.4.

2.3.5 Online Optimal Configuration Selection

To implement the classifier at the target platform, we need to do *only* the following:

1. Store the classifier parameters $\beta = [\beta_0, \beta_1, ..., \beta_N]$, where N is the number of input features ($N = 11$ in this work)
2. Implement Equation 2.3

At runtime, we read the input features using the PAPI calls for each workload snippet. Then, we plug these features and the β values to Equation 2.3, as shown in Figure 2.4. This gives the conditional probability of the occurrence of a Pareto-optimal configuration c_k^* given the input features $\mathbf{x}_k$. Then, the Pareto-optimal configuration c_k^* with the *maximum* conditional probability is selected as the output of the controller.

The proposed approach is highly scalable as it requires only a look-up table for a small number model coefficients β stored in the platform. This occupies very small storage space of only 282 bytes in the Odroid XU3 platform for the 11 features used in our work. Even if we store classifiers for multiple objective functions, such as energy, energy-delay product and performance, the file size does not exceed 2 kB. In general, the number of inputs to the classifier are always much smaller than the number of applications, phases and configurations. For example, if the system that needs to be optimized has hundreds of CPU cores, the proposed technique will still require to store only tens of model coefficients for any optimization objective. Note that when the number of features N becomes too large, the cost of computing the logistic function in Equation 2.3 can increase. In such cases, it is desirable to reduce and select the appropriate number of features using subset selection or Lasso regression [68]. Our approach is also general enough to consider more than two core types. In this case,

the characterization data has to include new types of cores. When the number of configurations grow, a subset can be characterized, as detailed in Section 2.3.3. Since the optimal classifier is designed offline, current offline computing power and existing classification algorithms can easily support solving iterative optimization techniques with tens of types of classes. Finally, the computation complexity of Equation 2.3 will not increase, making our approach scalable.

2.4 Experimental Results

This section first describes the experimental setup, including the details of the platform, benchmarks, baseline algorithms and the overhead of our approach. Then, we demonstrate the usefulness of the proposed dynamic Pareto-optimal configuration selection technique by comparing the results of the DyPO-Energy classifier with baseline algorithms and a recently proposed algorithm [1] running on the platform.

2.4.1 Experimental Setup

We present the experimental results performed on the Odroid XU3 platform running Ubuntu OS with kernel version 3.10 [51]. The platform is equipped with Exynos 5422 chip, which has four little (A7) cores and four big (A15) cores. The little core frequency can vary from 0.2 GHz to 1.4 GHz and big core frequency can change from 0.2 GHz to 2 GHz in steps of 0.1 GHz. The platform supports per cluster DVFS, i.e., the cores within the same cluster have to run at the same frequency and voltage. Changing the CPU cluster frequencies and setting of the core online and offline are supported in the platform using the cpu-freq driver. The platform also provides INA231 current monitoring sensors [139] that report the power consumptions for each CPU cluster, memory and GPU using the I2C driver. We set the sampling frequency of the current sensors to 5 ms to capture small transients in power consumption.

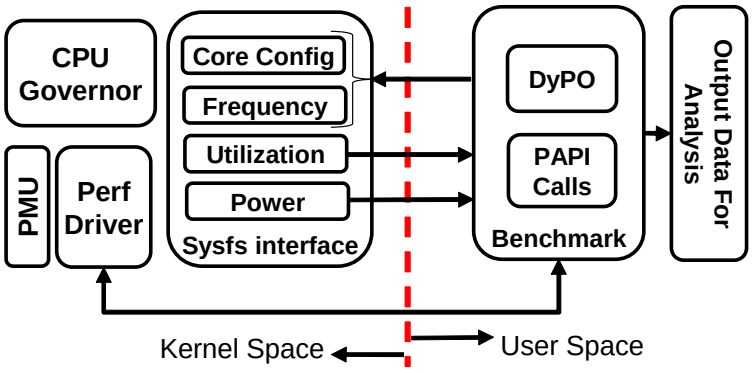

Figure 2.5: Implementation of DyPO in Linux Kernel 3.10.

Integration of the DyPO framework with the existing software infrastructure is shown in Figure 2.5. Our implementation is divided into the kernel space and user space. The *kernel space* contains the Perf driver and the CPU governors with a sysfs interface [89]. The Perf driver is mainly responsible for communicating with the ARM's performance monitoring unit (PMU) [18], which keeps track of different hardware and software counters. We enable the PMU to capture the performance counters listed in Table 2.1. We also utilize a custom CPU governor to capture per-core utilization through the sysfs interface. The *user space* contains the instrumented benchmarks with PAPI APIs that query the perf driver for performance counters [92]. At runtime, the hardware counters and CPU utilizations at each snippet of the application are used as inputs to the DyPO classifier. The classifier first finds the optimal frequency and core configuration, and then assigns them to the cores using the sysfs interface. We also export time stamps, classifier output and input features to a log file for debugging and offline analysis purposes.

Benchmarks: To validate our implementation, we use eighteen single- and multi-threaded benchmarks from MI-Bench [48], Cortex [138], and PARSEC [11] suites.

Default Governors: The Linux kernel implements a number of frequency governors that allow developers to optimize for a certain parameter. The *powersave* governor

runs the application at the lowest frequency such that the power consumption is minimized. The *ondemand* governor is used to meet a user defined utilization threshold by changing frequency [106]. The *interactive* governor is similar to the ondemand governor, except that it holds the frequency at a certain level for a fixed interval before making any changes. We compare our approach to these three governors [3] because they offer a wide variety of optimization goals and are implemented on millions of smartphones, making them competitive baselines [147].

Overhead Analysis: The DyPO framework induces instrumentation and algorithm runtime overheads. The instrumentation overhead can be measured in terms of the percentage of the extra instructions added to the benchmarks to log the performance counter data using the PAPI APIs. The baseline is the case when no APIs are inserted within the benchmark. As opposed to the baseline, the APIs in our approach have to be added in the source code to form different workload snippets, as explained in Section 2.3.2. We observe a very low mean and median overheads of 1.0% and 0.2% across all the 18 different benchmarks used in this chapter. The overhead of our runtime selection algorithm is 20μs, whereas the minimum and mean execution time of the workload snippets are 2.1 ms and 22.6 ms, respectively. That is, the runtime overhead of our approach is less than 1% of the smallest snippet and less than 0.1% of the mean value of the execution time of all the snippets. Our algorithm is called in the same way as the default frequency governor. As shown in Figure 2.5, the DyPO approach is implemented within the application to enable phase-level analysis. Therefore, during the decision process of the classifier, a single-threaded application pauses for 20μs. For multi-threaded applications, only one thread has to be paused for 20μs, other threads are not paused and continue to run normally.

[3] We kept the default cpuidle [105] governor active for the frequency governors to enable changes in the core configuration.

2.4.2 *Classifier Accuracy*

We use two classifiers in a hierarchical fashion, as explained in Section 2.3.5. The first classifier is a Level-1 classifier that outputs three probabilities. The highest probability class is chosen as the output of the classifier. Out of the three classes, two lead to specific frequency and core configurations. The third class fires another classifier, which we call the Level-2 classifier. The Level-2 classifier also outputs three classes that lead to specific frequency and core configurations.

The entire data set is divided into 60% training-validation set and 40% for test set on the actual platform. We train the classifiers using the training-validation set. Then, we use the classifiers at runtime for the entire data set (see Section 2.4.3 for results). Figure 2.6 shows the accuracy of both classifiers for the training-validation set. The accuracy for the Level-1 classifier across all the benchmarks is very high, with an average of 99.9%. The average accuracy of the Level-2 classifier for the benchmarks is 92.7%. The Blowfish benchmark never uses the Level-2 classifier, i.e., all of its snippets map to the Level-1 classifier only. Single-threaded applications achieve close to 100% accuracy for Level-2 classifier. However, the multi-threaded applications do not perform as well as the single-threaded benchmarks across all the three classes in the Level-2 classifier. For example, Blackscholes-4T shows 71% accuracy as opposed to 100% accuracy of the Basicmath application. This is because all the features of Blackscholes-4T are close to each other and harder to separate into different classes at the second level. We also assess the robustness of the classifiers to unknown data inputs by applying 5-fold cross-validation on the training-validation set. Our results for the 5-fold cross-validation show high average accuracy of 99.9% and 80.5% for the Level-1 and Level-2 classifiers, respectively.

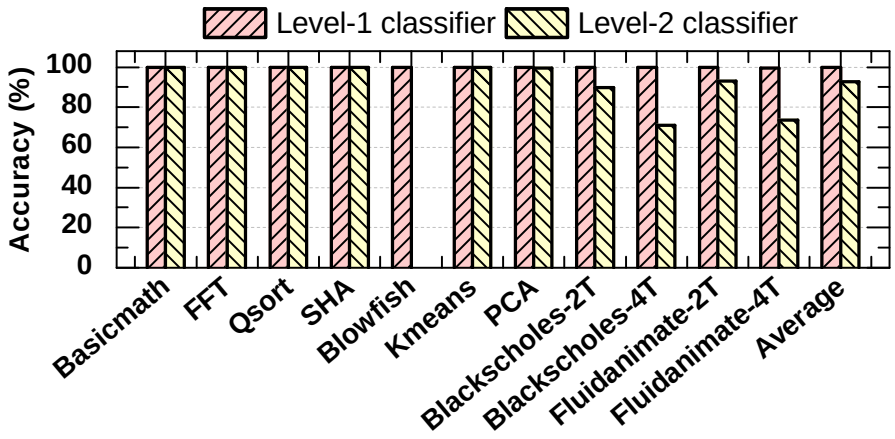

Figure 2.6: Accuracy of the two classifiers used on the Odroid platform. In multi-threaded benchmarks, -2T and -4T represents two and four threads, respectively.

2.4.3 Runtime Validation of DyPO

In this section, we present the validation of the proposed dynamic Pareto-optimal configuration selection approach by using the DyPO classifier at runtime. We use DyPO-Energy for illustration, since energy minimization is one of the main objectives in mobile platforms. At runtime, DyPO reads the hardware counters and utilization during each workload snippet as inputs to the classifier. Then, the classifier computes the probabilities of the optimal configurations using Equation 2.2. Finally, the configuration with the highest probability is assigned to the system for the next.

Figure 2.7 shows the comparison between offline characterized data for the entire application run at different frequency and core configurations ($\circ$), the Pareto-optimal points for power-execution time trade-off ($\diamondsuit$), the Pareto-optimal frontier for energy-execution time trade-off (—), powersave governor ($+$), interactive governor ($*$), ondemand governor ($\times$), and the proposed DyPO-Energy approach ($\triangle$). Since these plots show energy and execution time trade-off, the operating points closer to the Pareto-optimal frontier and low ordinate are desirable. The data points plotted using the

green markers (○) show the relative locations of the Pareto frontiers and the configuration space. This is useful in debugging and analyzing how different governor results get placed relative to these points. Figure 2.7(a) shows the results for the Basicmath application. The powersave governor lies to the extreme right of the plot at about 20 seconds execution time and consuming about 10 J of energy; this is expected as the goal of the powersave governor is to minimize power consumption. However, it does not minimize the energy consumption. In contrast, the DyPO-Energy approach runs the application at the lowest energy point of the Pareto frontier at about 14 seconds execution time and 8.7 J of energy consumption. It successfully achieves the energy minimization goal while also improving the execution time. Similarly, the DyPO-Energy approach leads to much lower energy consumption when compared with the interactive and ondemand governors. More precisely, the energy consumption is reduced by 42% (15 J to 8.7 J) and 46% (16 J to 8.7 J), respectively. This demonstrates the effectiveness of the DyPO technique in optimizing energy consumption. More importantly, none of the three default governors in the system lie on the Pareto-optimal point. In particular, the powersave and interactive governor are significantly off the Pareto curve. This is not desirable because there are clearly other configurations in the system that could have achieved lower energy consumption for the same execution time. The rest of the plots in Figure 2.7(b-n) show the energy consumption and performance trade-off for 13 more single-threaded applications. As expected, the interactive and ondemand governors consume significantly more energy, since they are optimizing the system to meet a utilization target. The powersave governor, on the other hand, does a good job in reducing the power consumption. However, it comes at the expense of performance and energy. In contrast, the results achieved by the proposed technique are always closest to the lowest point of the Pareto frontier for all applications.

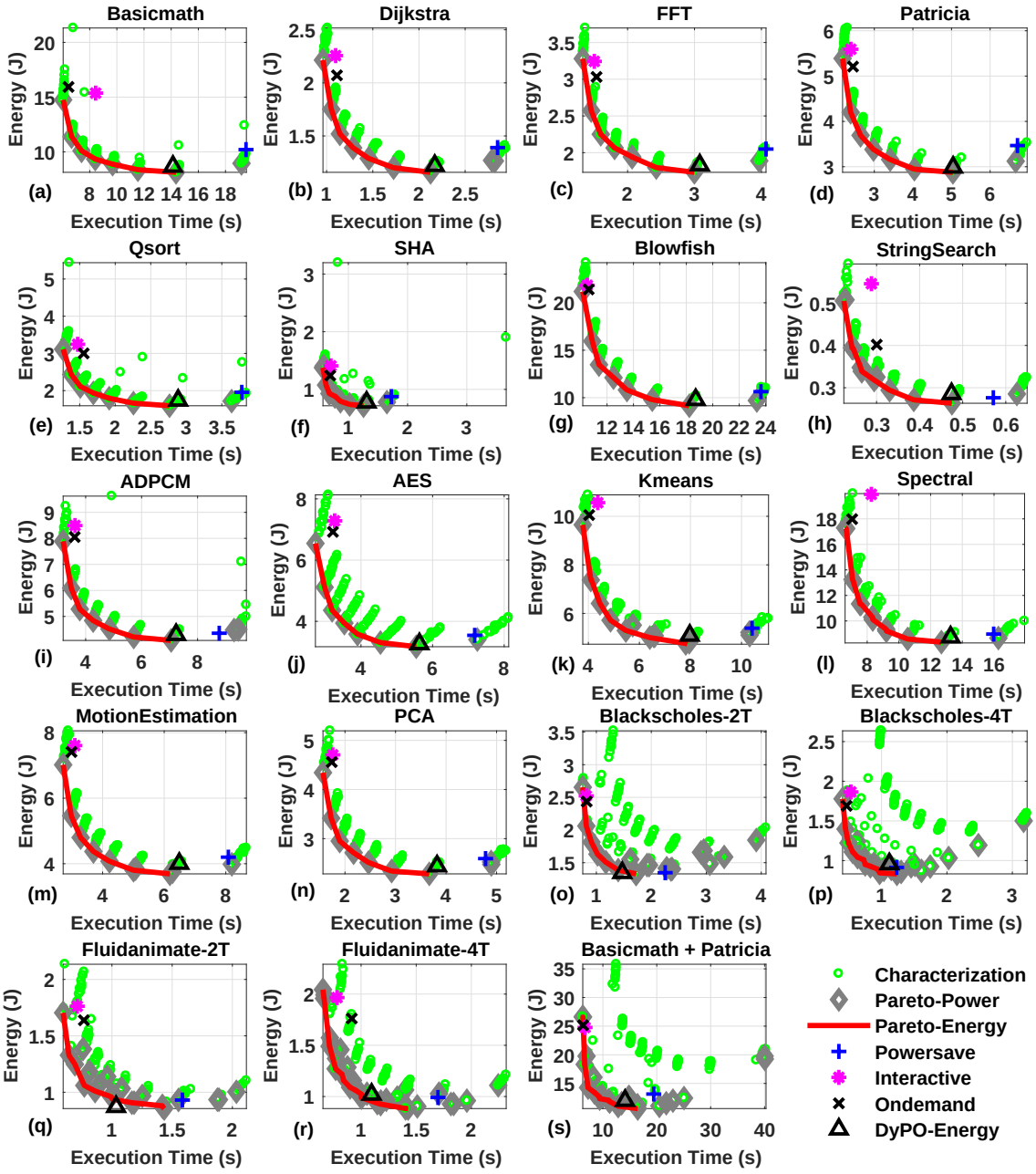

Figure 2.7: DyPO-Energy approach compared with the default governors running on the platform. In multi-threaded benchmarks, -2T and -4T represents two and four threads, respectively.

Multi-threaded Applications: As the complexity of mobile apps increases, it is also important to analyze the behavior when running multi-threaded applications. Therefore, we analyze their energy consumption and performance trade-off in Figure 2.7(o-r). In particular, Figure 2.7(q) shows the results obtained for the Fluidanimate application running with two threads. The DyPO-Energy approach lies below the Pareto-optimal curve, which means that our approach even outperformed the best case scenario of the characterization data, with a low energy consumption of 0.87 J and 1 second execution time. We observe that the lowest power configuration on the power and execution time Pareto curve (♢) leads to 2 seconds execution time. Moreover, it has substantially higher energy consumption compared to DyPO-Energy. This happens since the lowest power configuration utilizes fewer number of cores, which has a very large penalty when there are more than one active threads. Similarly, the Blackscholes application running with two and four threads and Fluidanimate application with four threads show that our technique achieves lower energy than the default governors, as illustrated in Figures 2.7(o)(p)(r). In these workloads, the DyPO-Energy moves up on the Pareto-optimal curve towards higher performance. This happens since the active threads increase the utilization, which demands a larger frequency. However, the proposed technique still stays at the Pareto frontier unlike the powersave, interactive and ondemand governors.

Concurrent Applications: The proposed runtime approach also works when multiple applications are running concurrently. More specifically, the instrumentation is specific to a particular foreground application. However, the classifiers operate on the performance counters, such as cache misses, non-cache external memory request, and number of active cores listed in Table 2.1. Therefore, when other background applications are running, the load perceived by the governor changes. For example, the background applications can increase the CPU utilizations, as well as hardware coun-

ters, such as LLC misses. Since the CPU utilization and hardware counters are inputs of the DyPO classifier, the proposed approach works with any number of applications and tasks running simultaneously with the foreground application. In fact, there were always hundreds of Linux OS background applications when we performed our experiments. To demonstrate the operation with multiple applications more explicitly, we simultaneously executed two applications, Basicmath (in foreground), and Patricia (in background). Figure 2.7(s) shows the results with this multiple application scenario. The proposed DyPO-Energy approach successfully minimizes the energy consumption compared to the default governors. More precisely, DyPO-Energy achieves 9% lower energy consumption, and at the same time, 27% faster execution time compared to the powersave governor. We also observe 52% lower energy consumption than the ondemand and interactive governors, albeit with a significant increase in execution time. This is expected since DyPO-Energy minimizes the energy consumption, while ondemand and interactive governors aim for performance. Most importantly, the optimal energy consumption of BML and Patricia running together is 12 J. This is almost the same as the sum of the individual optimal energy consumptions of BML and Patricia from Figure 2.7(a) and (d) equal to 11.7 J (sum of 8.7 J and 3 J). This further corroborates our claim that multiple applications can be optimized by using the DyPO-Energy approach effectively.

Note that we can choose any optimization objective in the DyPO technique, such as maximizing performance, minimizing energy with execution time constraint, minimizing the energy-delay product, as mentioned in Section 2.3.4. For example, we also experimented on performance (DyPO-Performance), in which case our framework always chose the highest points on the Pareto frontier (lowest execution time). This matches closely with the performance governor in the platform that is designed to achieve maximum performance. Also, the DyPO-PPW (maximizing performance per

watt) results are similar to DyPO-Energy in our setup, since the number of instructions are almost same for a given application run due to phase-level instrumentation.

2.4.4 Improvements in Energy and PPW

This section summarizes the advantages of the proposed methodology with respect to the default governors for each benchmark. To this end, we normalize the energy consumption, power consumption, execution time and PPW obtained for each governor with DyPO-Energy results. For example, Figure 2.8 shows the normalized energy consumption of all the benchmarks compared with the interactive, ondemand and powersave governors. We observe that the energy consumption reduces by 49% and 45% compared to the interactive and ondemand governor, respectively. For the interactive governor, even the smallest energy savings obtained by DyPO-Energy for the Basicmath application is 41%. The energy consumption achieved by the powersave governor is slightly more than 6% of the energy consumed by DyPO-Energy. Furthermore, this comes at the expense of almost 24% increase in execution time, as shown in Figure 2.9. The power consumed by the interactive and ondemand governors is about 3.5× that of the DyPO-Energy, as shown in Figure 2.10, while the power consumed by the powersave governor is about 23% lower. We also observe that the DyPO-Energy provides 93%, 81%, 6% more PPW than interactive, ondemand, and powersave governors, respectively (shown in Figure 2.11). Note that compared to the powersave governor, DyPO-Energy provides both energy savings and higher performance. When compared to the ondemand and interactive governors DyPO-Energy obtains substantial reductions in energy consumption albeit with lower performance, as shown in Figure 2.9. This is expected because the ondemand and interactive governors are designed for performance, not energy efficiency.

Comparison with Aalsaud et al. [1]: This section presents comparison of DyPO-

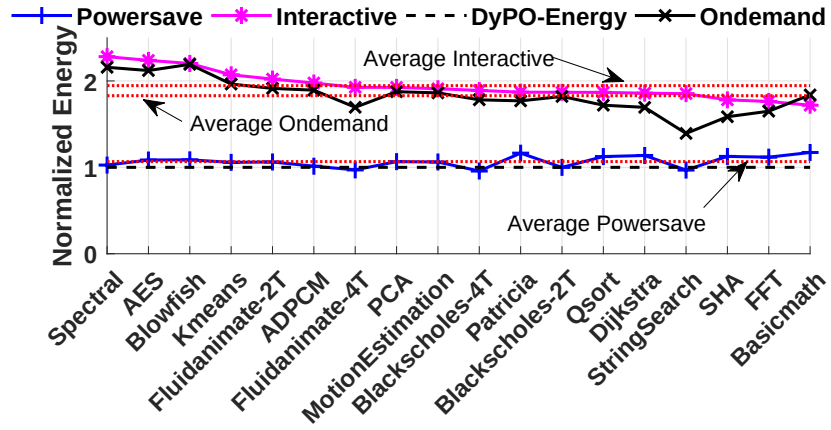

Figure 2.8: DyPO-Energy, Interactive, Ondemand and Powersave governor comparison for normalized energy consumption.

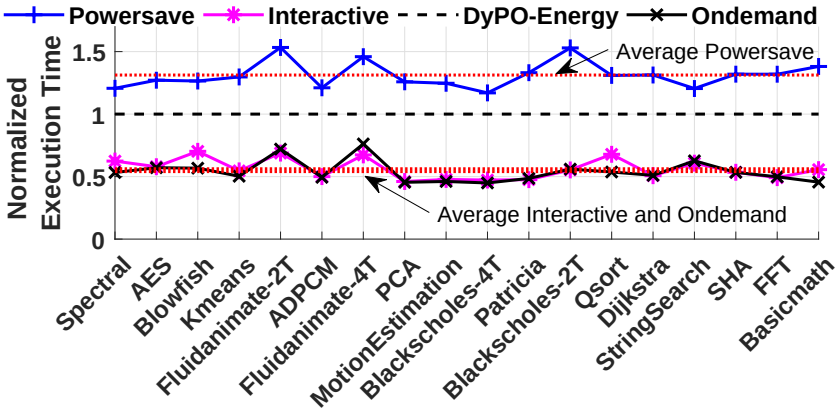

Figure 2.9: DyPO-Energy, Interactive, Ondemand and Powersave governor comparison for normalized execution time.

Energy against a state-of-the-art approach proposed by Aalsaud et al. [1]. They use power and performance (IPC: Instructions/Cycle) models that are linear functions of the number of little cores, big cores and one bias term. Each model is unique for an application and frequency level. That is, there are as many power and performance models as the number of supported frequencies in the platform for each application. These models are used for computing the PPW for all the supported frequencies and core configurations for a given application. There are two methods to their

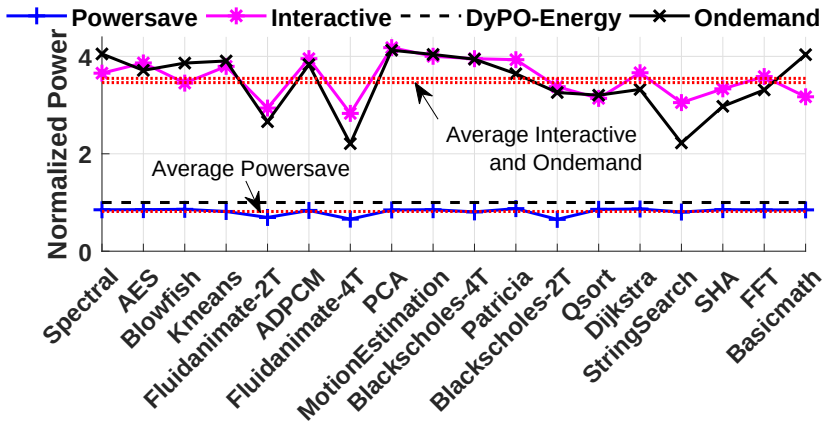

Figure 2.10: DyPO-Energy, Interactive, Ondemand and Powersave governor comparison for normalized power consumption.

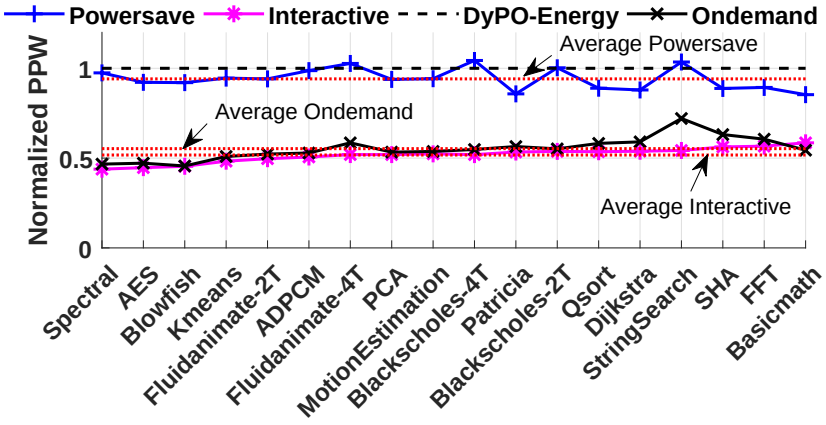

Figure 2.11: DyPO-Energy, Interactive, Ondemand and Powersave governor comparison for normalized PPW.

operation to maximize PPW at runtime. The first is offline (Aalsaud-offline) where the power consumption and performance models associated with an application are pre-characterized. The optimal configuration is found at runtime by a simple linear search through all possible frequency and core configurations. The second method is adaptive (Aalsaud-ADA) that works for an uncharacterized application. That is, an application for which the models are not known. Therefore, they determine the power and performance models at runtime for the adaptive method. To achieve this, they

first sweep the frequency every 200 ms. In each 200 ms interval, they measure power and IPC data for at least three different core configurations. Then, they apply linear regression on this data to find the models. Clearly, this is an overhead, since the system runs at non-optimal configuration for 200 ms times the number of frequency levels. However, this happens only one time, once the application is learned, the model is saved in a file for future use. Unlike the proposed approach, Aalsaud et al. [1] profiles the system at fixed time intervals. Since the PAPI APIs are not built to sample an application based on time, we used the perf utility [22] in the Odroid XU3 board to profile the applications every 50ms.

Figure 2.12 shows the PPW obtained by the DyPO-Energy, Aalsaud-offline and Aalsaud-ADA approaches normalized to the PPW obtained by running the ondemand governor. On average, the DyPO-Energy, Aalsaud-offline and Aalsaud-ADA provide 81%, 46% and 18% gain in PPW compared to the ondemand governor. Therefore, the DyPO-Energy approach shows 55% and 25% improvement in PPW compared to the Aalsaud-offline and Aalsaud-ADA approaches, respectively. Note that for applications Blackscholes-2T and String-Search, both Aalsaud-ADA and Aalsaud-offline perform worse than the ondemand governor. This is because for the String-Search application, the Aalsaud-offline approach used the configuration with a frequency of 1.2 GHz, and four little and big cores. This wastes the extra energy headroom, whereas the ondemand governor utilizes it by keeping the frequency below 1 GHz. We see similar behavior for the Blackscholes-2T application. In contrast, DyPO-Energy provides substantial gains in PPW compared to the approaches in Aalsaud et al. [1] and to the ondemand governor for all the benchmarks.

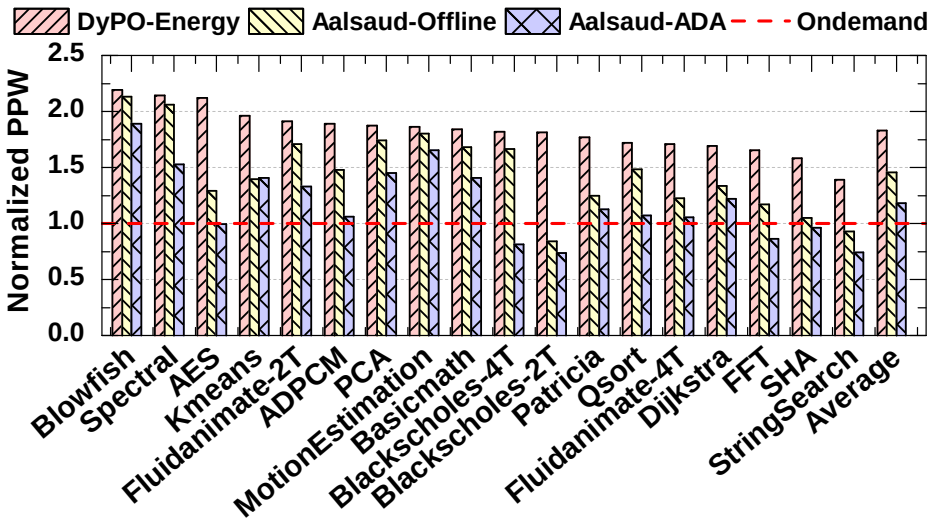

Figure 2.12: Comparison of the normalized PPW obtained using DyPO-Energy approach and Aalsaud et al. [1].

2.5 Conclusion

Continued demand for performance led to powerful mobile platforms with heterogeneous multiprocessor system on chips. These platforms provide many voltage-frequency levels and active core configurations that can be chosen at runtime. This chapter presented a novel methodology that finds the Pareto-optimal configurations at runtime as a function of the workload. The methodology consists of a combination of offline characterization and runtime classification. First, phase-level offline characterization for a number of benchmarks is performed to find the Pareto-optimal configurations for each workload snippet. Then, classifiers that map the characterized data to the Pareto-optimal configuration are learned offline using multinomial logistic regression. Finally, the classifiers are used at runtime to select the optimal configuration with respect to a specific metric, such as energy consumption. Our experiments show an average increase of 93%, 81% and 6% in performance per watt compared to the interactive, ondemand and powersave governors, respectively.

Chapter 3

DYNAMIC POWER BUDGETING FOR MOBILE SYSTEMS RUNNING GRAPHICS WORKLOADS

3.1 Introduction

Mobile platforms use system-on-chip (SoC) technology, which integrates specialized processing elements, such as the GPU, wireless modem, and DSP, in addition to the CPU cores. CPU cores do not dominate the power consumption under many application scenarios [14, 25]. For example, integrated GPUs have a relatively large surface area and can consume 5 to 10 times more power than the CPU cores, when running heavy graphic workloads. Furthermore, the total power consumption can fluctuate over time and exceed thermal power budget, as depicted in Figure 3.1a. For example, persistent violation of the power budget leads to thermal violations, while short peaks, such as the one at $t = 11$ s, are acceptable.

Thermal violations can have adverse effects on the device reliability and user experience [125, 129]. The power consumption needs to stay within a power budget to prevent thermal violations, as illustrated in Figure 3.1. As a result, allocating the power budget among the major platform resources and temperature control have become fundamental consideration for mobile platforms [1] . This can be achieved by putting a subset of the processing elements or shared resources to sleep states, or throttling their frequencies [16, 36, 120]. However, an ad hoc approach could easily cripple the performance, if it slows down the *performance-critical* processing element. For example, poor coordination between the CPU and GPU can easily

[1]For example, leading smart-phone manufacturers like Apple have a support page for customers to understand the thermal alleviation policies [6].

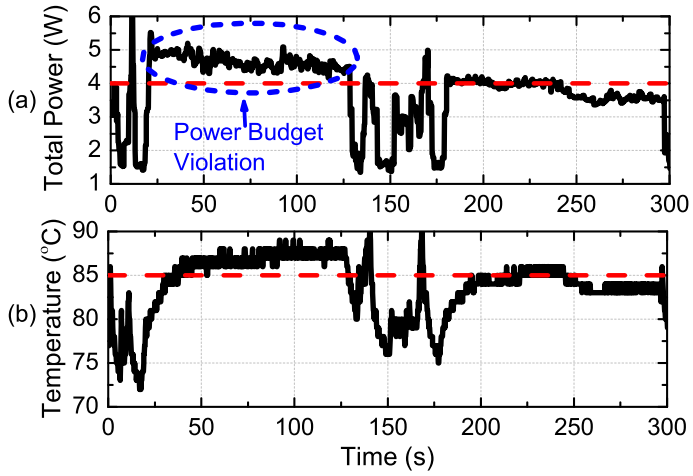

Figure 3.1: A sample of the total power consumption and CPU temperature while running the 3D-Mark application.

lead to a noticeable reduction in the frame rate, which would directly affect the user experience. Hence, there is a strong need for robust solutions to distribute the power budget efficiently among the active processing elements.

An ideal power budgeting approach would control every processing element in a coordinated manner, using dedicated control knobs. However, state-of-the-art mobile platforms traditionally offer fine-grained dynamic voltage-frequency scaling (DVFS) capability only for the CPU cores. Recently, this functionality has become available for the GPU [51, 64], but there is limited or no support for the rest of the processing elements. Furthermore, power consumption is dominated by the CPU cores and GPU when running graphics workloads Therefore, the rest of this chapter will focus on the CPU-GPU subsystem, even though the proposed approach is general for the whole SoC.

This chapter presents a power budgeting technique that allocates the power budget optimally between the CPU cores and GPU, while simultaneously adapting the achievable frame rate target. This is a challenging problem, since the CPU and GPU

utilizations vary dynamically as a function the workload. For example, when the application stage (*i.e.* processing points, evaluating different scenarios) takes more time, the system operation will critically depend on the CPU performance [2, 107]. This typically happens during physics simulations of realistic games. Similarly, higher complexity in the rasterizer stage (e.g., processing the pixels) makes the GPU the system performance bottleneck. When the GPU is the performance limiter, the CPU clock frequency can be lowered without any noticeable impact on the frame rate (and vice versa). However, both the CPU and GPU can become critical for certain periods of the application. As a result, it becomes crucial to determine by how much each resource should be throttled, especially when the platform is operating near the maximum power budget. To address this need, we present a mathematical model for the CPU-GPU subsystem power budgeting. We employ this model to determine the CPU and GPU clock frequencies that meets the target frame rate target subject to *dynamically varying* power budget constraints.

In general, an experimental evaluation provides the most accurate and decisive power and performance assessment. However, debugging and validating power management algorithms on a real world platform requires significant effort [21]. Moreover, validation of all features is not always feasible on a real hardware platform, since it requires the availability of a SoC, firmware, and an operating system, before testing new algorithms. Therefore, we performed *both experimental and simulation* studies. More precisely, we evaluated the proposed power allocation technique on a state-of-the-art mobile platform by running industrial benchmarks [137]. We also developed a trace-driven high-level simulator using Matlab/Simulink. This simulator enables us to debug, test and tune the power management algorithms before deploying them into target platforms. To ensure accuracy, we calibrated the simulator with measurements performed on an appropriate hardware platform [61].

The major contributions of this work are as follows:

- We propose a new power allocation technique, to compute *simultaneously* the *optimal* power budget allocations and the achievable frame rate under a power budget constraint.
- We present experimental evaluations on a commercial mobile platform using industrial benchmarks.
- We debug and validate the proposed approach on a wider range of scenario using a high-level simulator.

The rest of this chapter is organized as follows. Section 3.2 reviews the related work. Section 3.3 presents the proposed power budgeting technique. Section 3.4 discusses the experimental evaluation, and Section 3.5 concludes the chapter.

3.2 Related Research

Power consumption has remained as one of the most crucial design constraints for many years [93, 101, 117]. Traditionally, the peak power consumption has been a critical constraint for high-end systems [52, 76, 149]. With the advent of mobile devices, thermal-aware resource management [115, 122] and power budgeting [128] have become essential due to limited cooling options and demand for higher performance [99], respectively.

Several techniques have been successfully applied for power budgeting in multi-core systems [58, 66, 72, 102, 112, 150]. For example, the work presented in [112] compares different CPU power limiting techniques, such as DVFS, running average power limit (RAPL) [71], forced idleness [35] and thread packing [17]. Similarly, Reda et al. proposed an adaptive power capping technique that employs DVFS and adjusts the number of active cores [119]. The work presented in [76] uses a reactive dynamic

power partitioning algorithm to distribute the power budget unevenly among two CPU power domains to maximize instructions per second. Finally, power budget allocation per application instead of per CPU core was presented in [81]. Since these techniques target only CPUs, they cannot be applied to heterogeneous mobile systems running graphics workloads with complex CPU and GPU dynamics.

Graphics applications, such as Quake II are highly sensitive to DVFS [38]. This observation has drawn attention to mobile platform power management under graphics workloads. Most of the recent research is focused on minimizing the energy consumption, rather than power budgeting [15, 27, 108, 111, 128]. Furthermore, these approaches typically use heuristic governors, which require tuning to control the clock frequencies. For example, the technique presented in [27] relies on learning the performance model separately for each gaming workload. Similarly, Park et al. [27] proposed a heuristic technique that uses offline frequency tables for the CPU and GPU to minimize energy. Moreover, these models and tables have to be tailored to different hardware platforms. While the work presented in [70] employs a control-theoretic framework for managing the CPU and GPU clock frequencies to minimize energy, it *cannot guarantee* any power budget.

A number of power budgeting techniques have been proposed for heterogeneous MpSoCs. Singla et al. presented a dynamic thermal and power management algorithm that computes a total power budget using the predicted on-chip temperature [128]. If the predicted temperature exceed the maximum temperature threshold, their algorithm throttles first the frequency, and then the number of active processors until the predicted temperature drops below the threshold. Their work does not compute optimal power allocations for the CPU and the GPU, unlike the algorithm we propose in this chapter. Similarly, Wang et al. proposed a joint optimization technique for the workload and dynamic power budget distribution between the CPU

and GPU [143]. Their algorithm distributes the workload by assigning different data inputs for the same OpenCL compute kernels running in parallel on the CPU and GPU. This is different for graphics applications since the CPU has a fixed responsibility to execute application tasks, while the GPU has a fixed responsibility to execute rasterizer tasks.

In contrast to the previous approaches, the proposed technique is designed specifically for heterogeneous MpSoCs running demanding graphics applications under limited power budget. The proposed technique ensures that the total power consumption will stay within the power budget by computing simultaneously the optimal power budget allocations and the achievable frame rate.

3.3 Power Budget Allocation Mechanism

3.3.1 Preliminaries

This section presents the proposed power budget allocation technique. We limit the indices in the equations to a CPU cluster and GPU for the brevity of notation. However, our formulation can be generalized for any number of processing elements.

Power Budget: We define the power budget at time step k ($P^k_{\max}$) as the maximum allowable power consumption, which can be determined based on thermal constraints. We use a discrete time model since the control decisions in real systems are made at fixed control intervals. The power consumption of the CPU and GPU are denoted as p^k_{cpu} and p^k_{gpu}, respectively. Using this notation, we define the power slack at time step k as:

$$\Delta P^k_{\text{total}} = P^k_{\max} - (p^k_{\text{cpu}} + p^k_{\text{gpu}}) \tag{3.1}$$

When the total power consumption is greater than $P^k_{\max}$, *i.e.*, $\Delta P^k_{\text{total}} < 0$, the

total power consumption needs to be reduced to stay within the power budget. The CPU and GPU clock frequencies cannot be decreased arbitrarily, since this can cause an unnecessary loss in the frame rate. In contrast, $\Delta P^k_{\text{total}} > 0$ means that the current platform power consumption is less than the power budget. Therefore, the CPU and GPU clock frequencies could be increased without violating the power budget, if the frame rate is less than the target frame rate. However, an arbitrary increase in the clock frequency does not guarantee the best performance, and can lead to wasted power headroom. To formalize the power allocation problem, we express the total power slack as the sum of the change in the power consumption of the CPU (Δp^k_{cpu}) and GPU (Δp^k_{gpu}):

$$\Delta P^k_{\text{total}} = \Delta p^k_{\text{cpu}} + \Delta p^k_{\text{gpu}} \tag{3.2}$$

Our goal can now be expressed as determining Δp^k_{cpu} and Δp^k_{gpu} such that the frame rate target is met.

Performance Speedup: In the graphics pipeline, the CPU cores process batches, while the GPU processes frames, as illustrated in Figure 3.2. The effective frame processing rate (μ_d) is determined by the CPU (λ_{cpu}) and GPU (λ_{gpu}) throughputs, which are measured in *batches per second* and *frames per second*, respectively [2] . The rate at which the commands are fetched from the batch buffer is given by the ratio of the number of processed batches to the processing time. That is, a long batch processing time implies longer duration between two consecutive fetches, and leads to a smaller rate. Similarly, certain frames may consist of multiple batches. This is captured in our approach by the job ratio (r), which gives the number of batches per frame.

[2] We call μ_d as the frame rate for short. It is the fastest rate at which frames can be delivered to the display. Hence, it puts an upper bound on the display refresh rate.

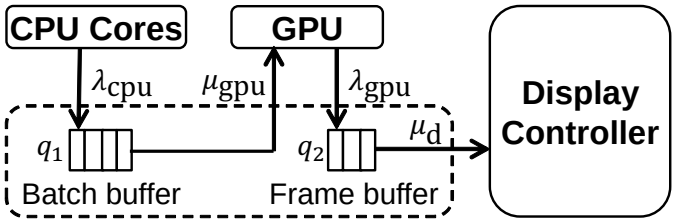

Figure 3.2: Example of a CPU-GPU queueing model showing batch buffer and frame buffer.

In order to quantify the impact of the CPU and GPU frequencies on the frame rate, we need to model how their respective throughputs change with frequency. Let f^k_{cpu} and f^k_{gpu} denote the CPU and GPU frequencies in time step k. Suppose that the ratio of the total processing time spent in the CPU pipeline, i.e, the CPU scalability factor [7], is given by x_{cpu}. Similarly, we denote the GPU scalability factor by x_{gpu}. We can use Amdahl's law [3] to express the throughput speedup that can be achieved by scaling the CPU frequency (S^k_{cpu}) and the GPU frequency (S^k_{gpu}) as:

$$S^k_i = \frac{\lambda^k_i}{\lambda^{k-1}_i} = \frac{1}{1 - x_i\left(1 - \frac{f^{k-1}_i}{f^k_i}\right)},\ i \in \{cpu, gpu\} \tag{3.3}$$

Since the scalability factor x in Equation 3.3 changes dynamically, we predict it at runtime using a linear function of individual hardware counters, their products and quotients. We estimated the coefficients of the function using offline linear regression [70]. Note that Equations 3.1–3.3 can be easily generalized to an arbitrary number of processing elements, as mentioned earlier. In the rest of the chapter, we use notation i to refer to the CPU and GPU.

A request to change the speedup can be triggered either due to violations of the power constraint, or a change in the frame rate target. For example, it may be necessary to slow down the clock frequency due to a negative power slack. Similarly, we may want to increase the clock frequency, when the power slack is positive and

the frame rate is below the target. Hence, we express the change in the speedup as:

$$\Delta S_i^k = \Delta S_{i,\mu_d}^k + \Delta S_{i,p}^k \tag{3.4}$$

where $\Delta S_{i,\mu_d}^k$ and $\Delta S_{i,p}^k$ denote the *change in speedup* due to the frame rate and the power budget, respectively. These terms can be obtained by applying the first order Taylor series approximation as follows:

$$\Delta S_{i,\mu_d}^k = \frac{\partial S_i}{\partial \mu_d}\Big|_k \Delta \mu_d^k \quad \text{and} \quad \Delta S_{i,p}^k = \frac{\partial S_i}{\partial p_i}\Big|_k \Delta p_i^k \tag{3.5}$$

The speedup definitions and list of the other parameters are summarized in Table 3.1.

3.3.2 Power Budget Allocation

We use Equations 3.2 through 3.5 to find the power consumption allocations and the required change in the frame rate target. To achieve this, we form step by step a system of equations as follows. Equation 3.2 gives the constraint on the total power slack, which would be distributed between the CPU and GPU. Substituting Equation 3.5 into Equation 3.4 gives two speedup equations, one for the CPU and the other for GPU. These equations can be written as:

$$\underbrace{\begin{bmatrix} \Delta P_{\text{total}}^k \\ \Delta S_{\text{cpu}}^k \\ \Delta S_{\text{gpu}}^k \end{bmatrix}}_{\mathbf{b}^k} = \underbrace{\begin{bmatrix} 1 & 1 & 0 \\ \frac{\partial S_{\text{cpu}}}{\partial p_{\text{cpu}}}\Big|_k & 0 & \frac{\partial S_{\text{cpu}}}{\partial \mu_d}\Big|_k \\ 0 & \frac{\partial S_{\text{gpu}}}{\partial p_{\text{gpu}}}\Big|_k & \frac{\partial S_{\text{gpu}}}{\partial \mu_d}\Big|_k \end{bmatrix}}_{\mathbf{A}^k} \begin{bmatrix} \Delta p_{\text{cpu}}^k \\ \Delta p_{\text{gpu}}^k \\ \Delta \mu_d^k \end{bmatrix} \tag{3.6}$$

Once the parameters in $\mathbf{b}^k$ and $\mathbf{A}^k$ are computed (as illustrated in Section 3.3.3), the unknowns, *i.e.*, the required change in CPU power Δp_{cpu}^k, in GPU power Δp_{gpu}^k, and frame rate target can be found by solving Equation 3.6 when the determinant of $\mathbf{A}^k$ is nonzero:

$$det(\mathbf{A}^k) = -\frac{\partial S_{\text{cpu}}}{\partial \mu_d}\Big|_k \frac{\partial S_{\text{gpu}}}{\partial p_{\text{gpu}}}\Big|_k - \frac{\partial S_{\text{cpu}}}{\partial p_{\text{cpu}}}\Big|_k \frac{\partial S_{\text{gpu}}}{\partial \mu_d}\Big|_k \neq 0 \tag{3.7}$$

Corner cases: We note that $det(\mathbf{A}^k)$ could be zero under four corner cases. For example, $det(\mathbf{A}^k) = 0$ when both $\partial S_{\text{cpu}}/\partial \mu_d = 0$ and $\partial S_{\text{cpu}}/\partial p_{\text{cpu}} = 0$. This condition means that the CPU speedup does not change either with allocated power or frame rate. Therefore, if this condition occurs, the remaining power budget should be allocated completely to the GPU. In general, when the speedup of one of the resources is oblivious to allocated power, one should allocate the extra power to the other processing element. The second corner case occurs for $\partial S_{\text{cpu}}/\partial \mu_d = 0$ and $\partial S_{\text{gpu}}/\partial \mu_d = 0$. That is, the speedup of neither of the processing elements depends on the frame rate change. If this condition happens, we allocate the power proportional to the derivative of their speed up with respect to the allocated power. Similarly, if neither speedup depends on the power allocation ($\partial S_{\text{cpu}}/\partial p_{\text{cpu}} = 0$ and $\partial S_{\text{gpu}}/\partial p_{\text{gpu}} = 0$), there is no need to allocate more power to any resource. The final corner case appears, if $\partial \mu_d/\partial p_{gpu}$ and $\partial \mu_d/\partial p_{cpu}$ have opposite signs. However, this can occur only if allocating more power, to the CPU or GPU, decreases the frame rate. If this unlikely scenario ever occurred, all of the power slack can be allocated to the processing element that would increase the frame rate.

3.3.3 Illustration of the Power Allocation Technique

The proposed power allocation technique can be used in conjunction with any control algorithm for which the parameters in Equation 3.6 can be expressed. Without loss of generality, we illustrate our technique using the state-space controller presented in [70]. Note that this controller alone can neither allocate the power budget optimally between a CPU and GPU, nor guarantee a power budget, *unlike the current work.*

We chose this controller for illustration, since it also uses the queuing model shown in Figure 3.2, where the CPU and GPU throughputs are shown as injections rates λ_{cpu} and λ_{gpu}. The ejection rate from the batch buffer to GPU is given as μ_{gpu}, while the ejection rate from the frame buffer to display is given as μ_d. If we denote the length of the control interval as T, the occupancy of batch buffer (q_1) and frame buffer (q_2) can be written as:

$$\begin{bmatrix} q_1^{k+1} \\ \\ q_2^{k+1} \end{bmatrix} = \begin{bmatrix} q_1^{k} \\ \\ q_2^{k} \end{bmatrix} + T \begin{bmatrix} \lambda_{\text{cpu}}^{k-1} & -r^{k-1}\lambda_{\text{gpu}}^{k-1} \\ \\ 0 & \lambda_{\text{gpu}}^{k-1} \end{bmatrix} \begin{bmatrix} S_{\text{cpu}}^{k} \\ \\ S_{\text{gpu}}^{k} \end{bmatrix} - T \begin{bmatrix} 0 \\ \\ \mu_d^k \end{bmatrix} \tag{3.8}$$

where r^k gives the average number of batches per frame in control interval k. The control output $\mathbf{S}^k$ can be found by applying a state feedback $\mathbf{G}^k(\mathbf{q}^k - \mathbf{q}_{\text{ref}})$, where $\mathbf{G}^k$ is the controller gain matrix and $\mathbf{q}_{\text{ref}}$ is the reference queue utilization:

$$\mathbf{S}^k = -\mathbf{G}^k(\mathbf{q}^k - \mathbf{q}_{\text{ref}}) + \begin{bmatrix} \frac{r^{k-1}\mu_d^k}{\lambda_{\text{cpu}}^{k-1}} \\ \\ \frac{\mu_d^k}{\lambda_{\text{gpu}}^{k-1}} \end{bmatrix} \tag{3.9}$$

Since the proposed budget allocation technique is not specific to this controller, we refer the reader to [70] for the details of the controller design. Next, we show the derivation of the parameters in Equation 3.6.

The left-hand side of Equation 3.6 ($\mathbf{b}^k$): The total power slack $\Delta P_{\text{total}}^k$ is computed using Equation 3.1. To find the speedup of the CPU and GPU throughput (ΔS_{cpu}^k, ΔS_{gpu}^k), we use the speedup expression given by Equation 3.3. That is,

$$\Delta S_i^k \triangleq S_i^k - 1 = \frac{\lambda_i^k - \lambda_i^{k-1}}{\lambda_i^{k-1}} \tag{3.10}$$

Derivatives with respect to frame rate in $\mathbf{A}^k$: We can find the first order derivatives of speedup, with respect to frame rate using Equation 3.9 of the feedback controller:

$$\frac{\partial S_{\text{cpu}}}{\partial \mu_d}\Big|_k = \frac{r^{k-1}}{\lambda_{\text{cpu}}^{k-1}}, \text{ and } \frac{\partial S_{\text{gpu}}}{\partial \mu_d}\Big|_k = \frac{1}{\lambda_{\text{gpu}}^{k-1}} \tag{3.11}$$

Derivatives with respect to power slack in $\mathbf{A}^k$: The power consumption of the CPU and GPU cores can be written as the sum of dynamic and leakage power. Since voltage typically scales linearly with frequency, power consumption during the control interval k can be written as a cubic polynomial in frequency [42, 153]:

$$p_i^k = a_i^k (f_i^k)^3 + b_i^k (f_i^k)^2 + c_i^k f_i^k + d_i^k \tag{3.12}$$

where, the coefficients $a_i,\ b_i,\ c_i,$ and d_i are the functions of hardware performance counters that change with time, to account for the workload dependent activity of the circuits. These parameters are characterized by measuring the power consumption offline, and fitting it to the model given in Equation 3.12. The maximum error in our power prediction was 8.2% for all benchmarks used in this chapter.

The derivative of speedup, with respect to power of a resource, can be expressed as:

$$\frac{\partial S_i}{\partial p_i} = \frac{\partial S_i}{\partial f_i}\frac{df_i}{dp_i} \tag{3.13}$$

$\partial S_i/\partial f_i$ is computed using the speedup Equation 3.3, while df_i/dp_i can be obtained from the power model given in Equation 3.12.

The unknowns in Equation 3.6: After $\mathbf{b}^k$ and the derivatives in matrix $\mathbf{A}^k$ are found, the unknowns (*i.e.* the optimal power allocations and frame rate adjustments, are given by $inv(\mathbf{A}^k)\mathbf{b}^k$.

3.3.4 Summary of Overall Operation

Figure 3.3 summarizes the proposed power budgeting technique. At the start of each control interval, we calculate the power slack $\Delta P^k_{\text{total}}$ using Equation 3.1. Then, we solve Equation 3.6 using the inputs from the frequency controller, as explained in Section 3.3.3. This gives the required change in frame rate target $\Delta\mu^k_d$, as well as the required change in the CPU and GPU power. Finally, we use $\Delta\mu^k_d$ to update the frame rate target as:

$$\mu^{k+1}_d = \mu^k_d - \Delta\mu^k_d \tag{3.14}$$

To ensure convergence to the target power budget, we apply an iterative linear search for the change in frame rate target $\Delta\mu^k_d$ in steps of 1 FPS until power budget constraint is met. This value is then used in Equation 3.9 to find the required speedups S^k_{cpu} and S^k_{gpu}. Finally, we utilize Equation 3.3 to calculate the actual CPU and GPU clock frequencies, given S^k_{cpu} and S^k_{gpu} as follows:

$$f^k_i = \frac{f^{k-1}_i}{1 - \frac{1}{x_i}\left(1 - \frac{1}{S^k_i}\right)} \tag{3.15}$$

3.4 Experiment and Simulation Results

In this section, we present the hardware platform setup used to evaluate the proposed power budgeting technique. Then, we discuss the experiment results. Finally, we describe our high-level simulations used to validate the proposed power budgeting algorithm.

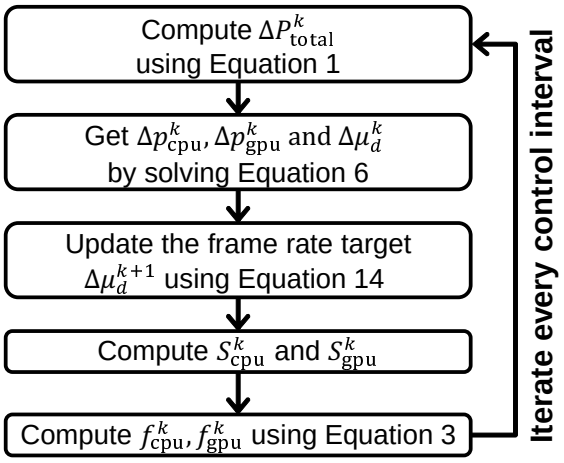

Figure 3.3: Summary of the power budgeting technique, showing the steps in each control interval.

3.4.1 Hardware Experimental Setup

We implemented our technique on a quad-core Atom$Z3775$ [62] based platform, shown in Figure 3.4. The platform runs Android JellyBean 4.2.2 [37]. The Atom chip consists of Intel HD graphics core with 4 execution units that operate in the frequency range of 244 MHz to 778 MHz. There are four CPU cores whose operating frequency ranges from 533 MHz to 2192 MHz.

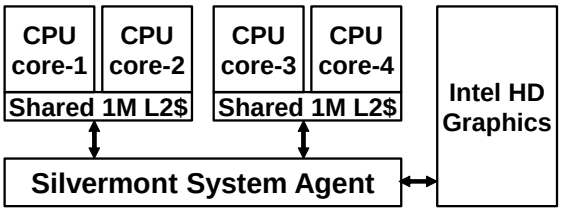

Figure 3.4: Block diagram of the Atom chip [137] used in our experiments.

The CPU-GPU queueing model shown in Figure 3.2 is valid for both ARM and x86 based MpSoCs. Therefore, the proposed technique can also be applied to other platforms with integrated GPUs. Successful implementation requires instrumenting

the Android kernel to obtain the CPU and GPU frequencies, frame rate, the occupancies, injection rates and ejection rates of the batch and frame buffers. In addition, the CPU and GPU power consumptions need to be read through a sensor, or a power meter. In particular, we used a power meter similar to Trepn profiler from Qualcomm [116]. Furthermore, instrumenting of the batch and frame buffers to obtain the injection and ejection rates is nontrivial. To achieve this, we identified the display kernel functions that are called whenever a frame is written to the frame buffer using the debug log. Then, we added global counters that can be read through the sysfs interface [89] in every control interval. The ratio of the number of frames to the length of the control interval—in our case, 50 ms—gives the frame rate.

Our implementation is partitioned between the kernel space (within the CPU and GPU drivers) and user space (proposed power budgeting technique). The maximum achievable frame rate is limited by the display refresh rate. For example, our experimental platform supports a maximum frame rate of 60 FPS. The proposed power allocation technique targets the operation regions where the power budget forces the achievable frame rate less than or equal to this value. The proposed technique is invoked at every 50 ms to allocate the power budget and control the clock frequencies. The control interval is set as 50 ms, since this causes negligible overhead, and at the same time allows processing three frames assuming a frame rate of 60 FPS. Finally, we validated the power models using NI-USB 6289 data acquisition unit [97].

Benchmarks: We ran a set of representative graphics applications, such as 3D-Mark, GLbench, Citadel, Nenamark2 and Jet-Ski on the platform.

Power budget: $P_{\max}$ is usually determined by the power control unit using the thermal constraints and available battery level. Increasing power consumption leads to an increase in the temperature [130]. For example, as the total power consumption of our experimental platform increased from 2.1 W to 5.5 W, the temperature of the

heat sink rose from 39°C and 55°C. At any point in time, the difference between the maximum safe temperature and current temperature can be used to determine the power budget that can be allocated to the CPU and GPU [128]. Hence, the power budget can change dynamically during the runtime of an application to utilize the available thermal headroom. It has been also shown that computational sprinting can provide a significant performance gain by allowing the power consumption to exceed the thermal power budget for short durations of time [118]. Hence, one can allow short violations ($\sim$ 200 ms) when the frame rate needs to be boosted.

Many platforms have heuristic policies implemented in firmware to reduce the power when the total power exceeds P_{max}. In our setup, we allow P_{max} to change to any desired user-defined level to enable us to undertake controlled experiments. To accomplish this, we first obtained the total power consumed by the CPU and GPU for an unconstrained system P_{unconst}. Then, we chose a set of P_{max} values at 50%, 70%, 80%, and 90% of P_{unconst} in order to study the sensitivity of our technique to P_{max}.

Heuristics: We compared our technique to the default static and dynamic heuristic algorithms that distribute the power consumption between the CPU and graphics components such that the sum of the individual power consumptions is constant [56].

a) Static: The CPU and GPU power budgets have fixed ratios during the lifetime of the system (typically 90% of P_{max} is assigned to the GPU, and 10% is assigned to the CPU cores).

b) Dynamic: The CPU and GPU power budgets are distributed proportionally to a weight parameter that signifies the criticality of the GPU. This weight is incremented or decremented dynamically as a function of the GPU utilization, which is defined as the ratio of the GPU active time to the control period. As a result, larger GPU

utilization leads to larger power budget allocated to the GPU. Likewise, more power is allocated to the CPU as the utilization of the GPU reduces. The utilization thresholds and power increment/decrement step size are fixed, and are tuned for a given platform.

3.4.2 Experimental Results on the Hardware Platform

Power consumption evaluation: Figure 3.5 shows the sum of the CPU and GPU power consumption of 3D-Mark benchmark over 15 seconds. During the first 5 seconds, the power budget is set to 50% of the unconstrained power consumption. We observe that the proposed technique successfully maintains the power consumption within the budget. The power budget is then raised to 80% of the unconstrained power consumption during the following 5 seconds. The controller responds quickly by increasing the CPU and GPU clock frequencies. Similarly, the power consumption is throttled at $t = 10$ second, immediately after lowering the power budget again to 50%. This shows the robustness of the proposed approach in meeting the power budget target. A more detailed analysis of the data also reveals that the total power can occasionally deviate from the power budget. For example, we observe spikes around $t = 1.5$, $t = 3.5$, and $t = 14$ seconds in Figure 5. These spikes can occur at runtime due to the change in the workload and quantized values of p-states for the CPU and GPU. Figure 6 shows the mean absolute percentage error between the target power budget and the achieved power consumption using the proposed approach for the benchmarks running with $P_{\max}$ values at 50%, 70%, 80%, and 90% of P_{unconst}. The error across the benchmarks is less than 6% indicating the proposed power budgeting technique successfully meets the target power consumption.

Next, we analyze the CPU-GPU power budget distribution at two different power budget settings. One setting is tight ($P_{\max}$ is 50% of the unconstrained power), and the other is loose ($P_{\max}$ is 90% of the unconstrained power). Figure 3.7 shows that

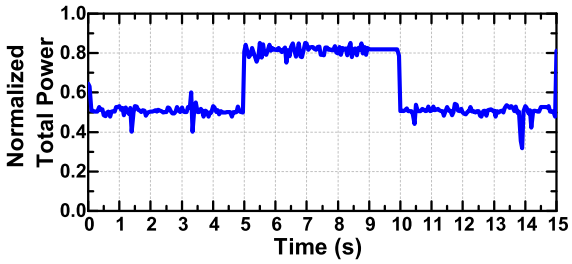

Figure 3.5: The sum of the CPU and GPU power consumption for 3D-Mark benchmark showing two levels of power budget. The trace is 15 seconds long, *i.e.*, 300 control intervals.

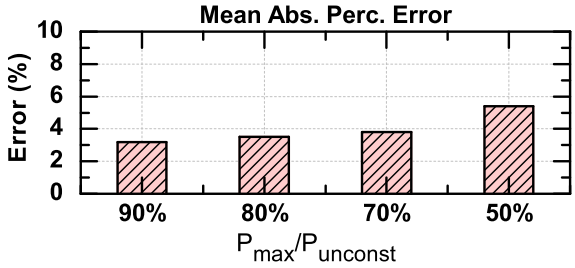

Figure 3.6: Deviation from the power budget constraint for different values of the power budget. Each bar reports the average of the error in 3D-Mark, GLbench-Egypt, Citadel, Nenamark2 and Jet-Ski benchmarks.

the average CPU power hardly reaches 10% of the total power consumption for our benchmarks. This is expected as most graphics applications are GPU heavy. We also note that the CPU power consumption varies across different benchmarks. In particular, the Jet-Ski game has higher fraction of CPU power consumption than the rest of the benchmarks. This shows the importance of adapting the power budget dynamically. Finally, the ratio of the CPU to GPU power consumption changes as a function of the power budget. This indicates that static allocation would either over- or under-utilize the power budget, while a dynamic allocation has a potential to adapt to different workloads.

Performance evaluation: Figure 3.8 compares the frame rate achieved with the

proposed power budgeting technique against the heuristic algorithms under different power budget scenarios [3] . Note that the heuristics in this chapter were highly tuned. Thus, we are comparing our technique's results against competitive baseline algorithms.

When the power budget is set to 50% of the unconstrained power consumption, the proposed technique significantly outperforms the heuristics, as shown in Figure 3.8a. In particular, when running the Glbench-Trex benchmark, the proposed technique achieves 38% and 64% higher frame rate than the static and dynamic heuristics, respectively. On average, the proposed technique delivers 15% higher frame rate than the static heuristic, and 10% higher frame rate than the dynamic heuristic. These improvements are achieved because our solution controls the CPU and GPU clock frequencies very effectively under tight power budget constraints. When the power budget is relaxed (as shown in figures 3.8b, 3.8c, and 3.8d) our technique still outperforms the heuristic algorithms, albeit with a smaller gain in frame rate. It is

[3]For fairness, all three algorithms were evaluated under the same power consumption in any given scenario.

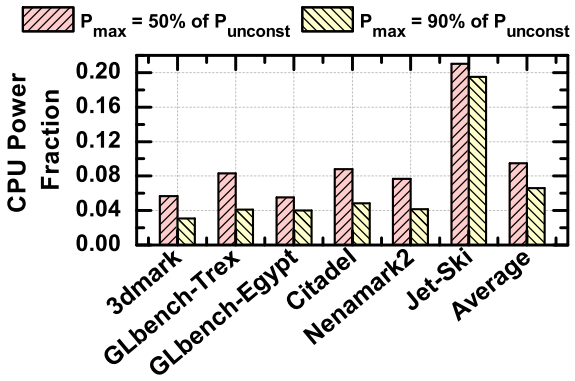

Figure 3.7: Experimental results for two different power budget values (50% and 90% of the unconstrained power P_{unconst}). The CPU power fraction of the power budgets is plotted for each of the benchmarks. The GPU power fraction is equal to (1− CPU power fraction).

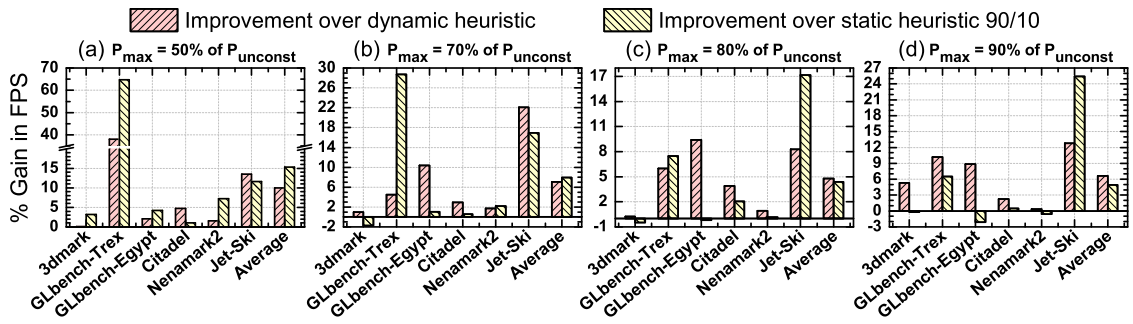

Figure 3.8: Comparison of the throughput gain (FPS) achieved with the proposed technique with respect to 1) dynamic heuristic, and 2) static heuristic that allocates 90% of $P_{\max}$ to GPU and 10% of $P_{\max}$ to CPU.

also important to note that the proposed technique delivers a consistent performance across all the benchmarks, while there is no clear trend for the heuristics. For example, the dynamic heuristic performs better than the static heuristic for the GLbench-Trex benchmark, when the power budget was less than or equal to 80% of the unconstrained power consumption, as shown in figures 3.8a, 3.8b and 3.8c. However, when the power budget increases, the static heuristic starts to perform better, as shown in Figure 3.8d. We observe smaller frame rate improvement for Nenamark2 and 3D-Mark compared to others such as Jet-Ski. Due to the relatively lower frame complexities in Nenamark2, the heuristics are able to meet the frame rate target even with lower power budgets. Similarly, the heuristics are able to meet the frame rate target for 3D-Mark because the frame rate saturates quickly even with a high power budget. Hence, the proposed approach does not show significant improvement in these cases.

Finally, Figure 3.9 shows the average frame rate as a function of the power budget. When the power budget is as low as 50% of the unconstrained power consumption, both heuristics perform significantly worse than the proposed technique. The heuristics close the gap gradually and approach the performance of the proposed technique when the power budget relaxes to 80% of the unconstrained power consumption.

However, any further increase in the power budget degrades the performance of the heuristic algorithms. The mean and median values of the frame rate across all the benchmarks are 42 FPS and 49 FPS, respectively. The large values of the mean and median indicate that many applications achieved high frame rates. In fact, the maximum measured frame rate is 60 FPS, while the minimum observed value is 11 FPS, which occurred only once at the lowest power budget. In the figures, we report only the normalized values due to confidentiality reasons.

In conclusion, the proposed approach not only provides a high throughput, but also utilizes the available power slack more effectively. Hence, it achieves its goal of allocating the optimal power consumption to the CPU and GPU under a given power budget.

3.4.3 Simulation Framework

In this section, we first provide the motivation and details of our high level simulator and then present the simulation results.

Use of the Simulator: The ultimate test for power and performance validation is running the applications on a hardware platform, as presented in the previous section. However, validating power management algorithms, such as the proposed technique,

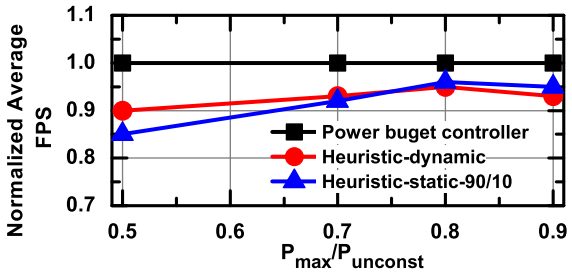

Figure 3.9: The average frame rate across all benchmarks for each of the power budget algorithms.

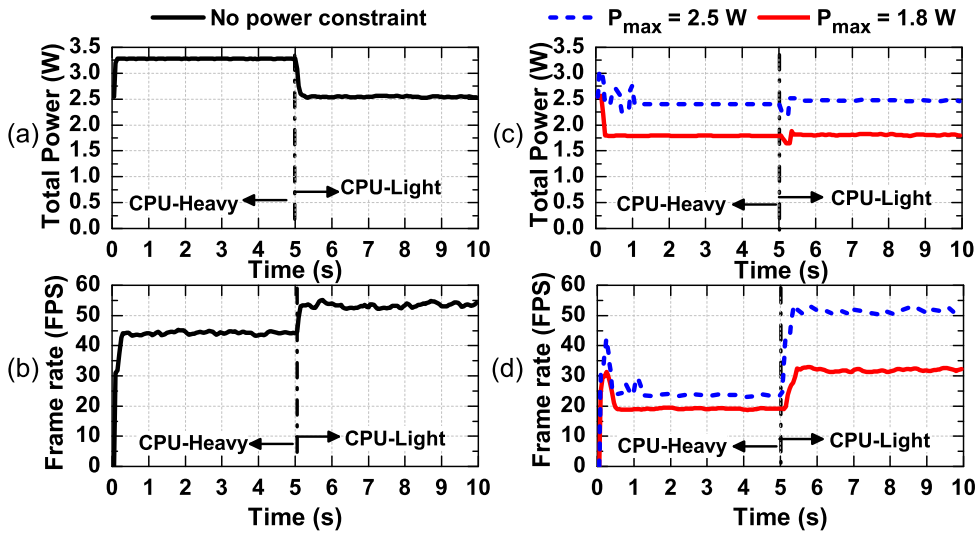

Figure 3.10: Simulation result of the proposed power budgeting technique showing the sum of the CPU and GPU power consumption and frame rates for different power budget values and workload phases. The GPU is under heavy load to simulate graphics intensive applications like gaming.

requires running minutes of workloads, which generally implies several thousands of frames. This range of runtime can be easily achieved on real platforms, but implementing and debugging the algorithm directly on the platform is time consuming. More specifically, limited observability at the kernel level makes debugging and tuning of the algorithms difficult. Furthermore, the compile time after each modification takes a significant amount time. As a result, the debugging and tuning time ends up in the order of weeks. The traditional cycle-accurate architectural level simulators, like gem5-gpu [114], are also not suitable for validating power management algorithms due to their long execution times. For example, simulating thousands of frames on gem5-gpu can easily take weeks to run. Therefore, we developed a trace-driven high-level simulator using Matlab/Simulink *to develop, debug and tune the algorithms*. After the algorithm is validated and tuned on the simulator, we port the code and

parameters to the real platform. This reduces the development effort on the platform to a few days.

Simulator Infrastructure: The simulator was built using the performance and power models for the CPU and GPU that were derived from the real experiments using Minnowboard [61] and Odroid-XU+E [98]. The CPU and GPU interactions were modeled using the queueing system shown in Figure 3.2. The simulator operates at the batch level, *i.e.*, the CPU and GPU read in batches from a trace file. Each batch comes with processing time, sleep time, frequency and utilization for both CPU and GPU cores. To obtain the input traces, we modified the Android kernel such that these values can be stored as a function of time. Then, we ran the target benchmarks on the platform to collect the reference input traces. The reference values from the trace are then used to simulate the performance and power under varying workload and control policies implemented in our simulator. To ensure good fidelity, we calibrated the simulator with the help of the workloads and algorithms implemented both in the hardware [61, 62] and the simulator. This resulted in less than 3% error in power consumption across all supported frequencies and 5% error in the frame rate. In addition, we also verified that the batch and frame processing times reported by the simulator match the measured values when the CPU and GPU frequencies were kept at their reference values. We note that this simulator does not provide visibility in terms of how each batch is composed, but it captures the impact of the CPU and GPU frequencies on the frame rate accurately. Moreover, operating at a high level enables us to evaluate power management algorithms while running thousands of frames at a speed of 60 frames/min.

3.4.4 Simulation Results

One of the main benefits of high-level simulation is the ability to test the power management algorithms under workloads that are difficult to generate on the target platform. To evaluate the power budgeting algorithms *under different corner cases*, we simulated a workload that consists of a {CPU heavy, GPU heavy} phase for the first 5 seconds, and a {CPU light, GPU heavy} phase for the next 5 seconds. Next, we provide the results for the proposed power budgeting algorithm and the static heuristic algorithm to present a deeper understanding of how each algorithm behaves under different corner cases.

Proposed Algorithm

Figure 3.10 shows the total power and frame rate for the 10 seconds simulation workload under different power budgets using the proposed algorithm.

{CPU heavy, GPU heavy} phase: Without any power constraint, the maximum power consumption is about 3.3 W, and the frame rate is 43 FPS, as shown in Figures 3.10a and 3.10b. The proposed technique successfully stabilizes the total power at 2.5 W and 1.8 W after constraining the power budget to about 80% and 60% of the unconstrained power, as depicted in Figure 3.10c. As a result of the reduced power budget, the frame rate drops to 24 FPS and 20 FPS, as presented in the CPU-heavy region of Figure 3.10d.

{CPU light, GPU heavy} phase: Lowering the CPU load immediately reduces total power consumption when there was no power constraint. This transition is clearly visible at time $t = 5$ second in Figure 3.10a. In contrast, when there is a power constraint, the power slack released by the CPU is allocated to the GPU. Hence, the total power consumption remains flat, as depicted in Figure 3.10c. In par-

ticular, when the power constraint is 2.5 W, the CPU power consumption drops from 0.8 W to 0.5 W after the workload changes, as detailed in Figure 3.11. The second consequence of lowering the CPU load is increased frame rate, as shown Figure 3.10b. Without the proposed power reallocation technique, the GPU would become the performance bottleneck and limit the frame rate. Our technique, however, allocates the resulting 0.3 W power slack to the GPU in less than 150 ms by increasing the GPU clock frequency. In turn, the GPU starts processing more frames within the same total power budget. Consequently, the frame rate increases to 51 FPS, as shown in Figure 3.10d. Similarly, the proposed technique successfully redistributes the power budget under a 1.8 W power constraint and achieves a frame rate of 31 FPS.

In summary, the proposed technique effectively redistributes the power budget and achieves a high frame rate. Figure 3.11 summarizes the precise distribution of the total power budget between the CPU cores and GPU for all the workload and power constraint scenarios considered in this simulation.

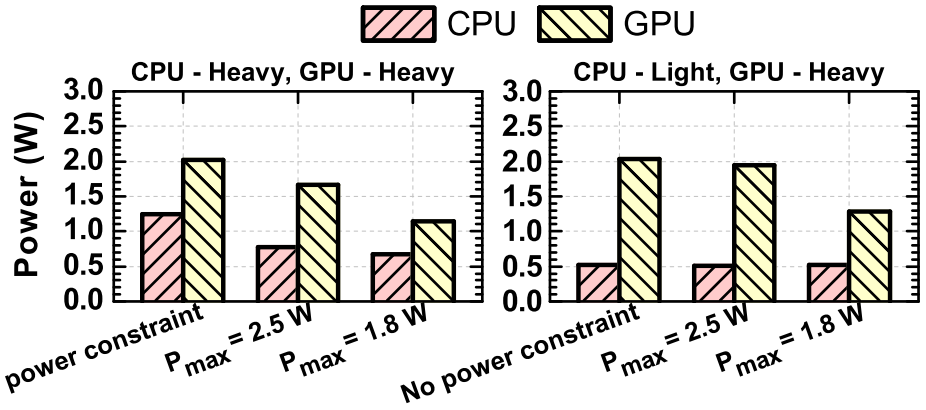

Figure 3.11: Power budget distribution between the CPU and GPU for the three power budget values and workloads.

Static Heuristic Algorithm

Another use of the simulator is the ability to test a wider set of scenarios, which would take days on the real platform. For example, the static heuristic we used in the experiments allocated 90% of the total power to the GPU and the remaining 10% to the CPU based on experience. To explore the impact of these allocations, we re-ran the same trace, that consists of a 5 second CPU-Heavy phase followed by a 5 second CPU-Light phase for three scenarios with 10%, 30%, and 50% CPU power allocations. Growing the CPU power allocation from 10% to 50% increases the frame from 19 FPS to 28 FPS during the CPU-Heavy phase, as shown in Figure 3.12b. However, this also reduces the frame rate by half during the CPU-Light phase, since GPU has to run at a lower frequency. Similarly, allocating 30% of the power budget marginally improves the frame rate during the CPU-Heavy phase, but significantly hurts it later. This shows that 90% GPU - 10% CPU allocation is relatively better than the other allocations. However, static allocation, by its nature, cannot adapt to workloads and meet time varying requirements, unlike the proposed technique.

3.5 Conclusion

Mobile platforms operate under tight power budgets due to limited cooling solutions. Therefore, it is critical to distribute the limited power budget efficiently between the GPU and CPU cores, when running graphics applications. In this chapter, we present a power budgeting technique, for the GPU-CPU subsystem, which does not require any tuning, unlike existing heuristics. Furthermore, the proposed technique not only provides high throughput, but also utilizes the available power slack more effectively. Therefore, it successfully achieves its goal of allocating the optimal power consumption to the CPU and GPU, under a given power budget. The

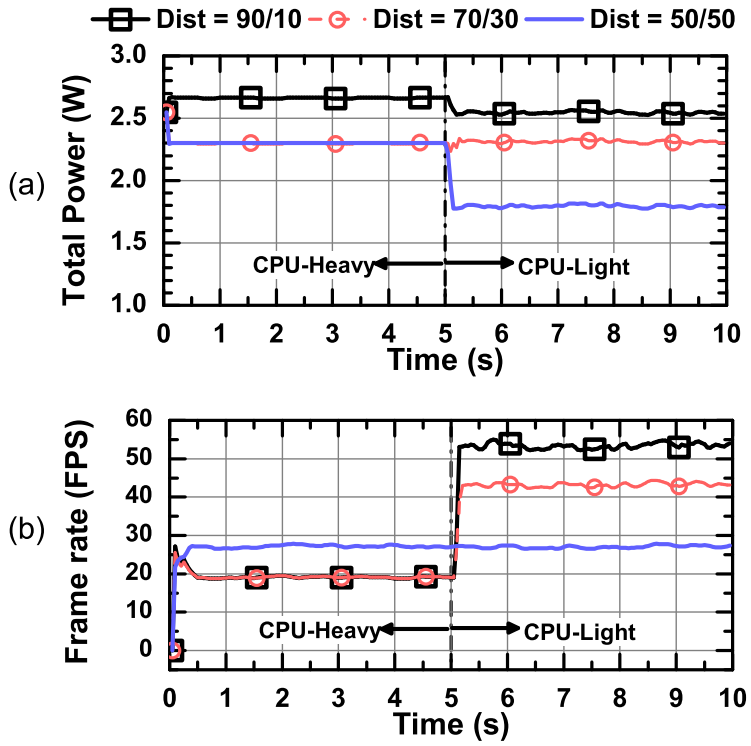

Figure 3.12: Static heuristic algorithm result for (a) the sum of the CPU and GPU power consumption and (b) performance for different distributions of GPU/CPU powers. The power budget is set to 2.5 W. Dist=Y/X in the legend indicates that Y% of 2.5 W is allocated to the GPU and X% of 2.5 W to the CPU.

effectiveness, of the proposed technique, has been demonstrated using both simulations and experiments. The experiments being performed on a state-of-the-art mobile platform running industrial benchmarks. In future, we plan to perform experimental evaluation of our power budgeting framework by adding more components of the MpSoCs, such as memory.

Table 3.1: Notation Table

Notation	Description
$P_{\max}$	Power budget: maximum allowable power consumed by the CPU and GPU
ΔP_{total}	Power slack
k	Time step
p_i	Power consumed by the CPU/GPU
λ_i	Throughput of the CPU/GPU
f_i	CPU/GPU frequency
x_i	Scalability factor of the CPU/GPU
S_i	CPU/GPU speedup in throughput
ΔS_i	CPU/GPU change in the speedup
$\Delta S_{i,\mu_d}$	CPU/GPU change in the speedup due to frame rate
$\Delta S_{i,p}$	CPU/GPU change in the speedup due to power consumption
μ_d	Frame processing rate
q_1	Queue occupancy of batch buffer
q_2	Queue occupancy of frame buffer
T	Length of each control interval (50 ms)
r	Average number of batches per frame
$\mathbf{q}_{\text{ref}}$	Column vector of reference queue utilizations for batch and frame buffers
$\mathbf{G}$	Controller gain matrix
P_{unconst}	Power consumed by the CPU and GPU combined for unconstrained case

Chapter 4

AN ONLINE LEARNING METHODOLOGY FOR PERFORMANCE MODELING OF GRAPHICS PROCESSORS

4.1 Introduction

Graphically-intensive mobile applications, such as games, constitute about 18% of the most popular smartphone application categories [5]. Consequently, integrated GPUs have become an indispensable component of mobile processors due to the increasing popularity of graphics applications. Our measurements show that the GPU power consumption accounts for more than 35% of application processor power when running many of these applications. The GPU frequency cannot be reduced arbitrarily to save power, since it also determines the achievable frame rate, which has a significant impact on the user experience. Therefore, there is a growing need to use graphics performance models that can accurately and judiciously control the GPU frequency.

The primary graphics performance metric is the number of frames that can be processed per second, since this limits the maximum display frame rate. Therefore, we use the time the GPU takes to process a frame as the performance metric. Frame time highly correlates with GPU frequency, and is dependent on the target application. Furthermore, it varies significantly throughout the lifetime of an application, as shown in Figure 4.1. That is, the frame time is a multivariate function of the frequency and workload, where the latter is captured by the performance counters. Therefore, an effective GPU performance model must adapt to the dynamic workload variations to accurately predict the frame time as a function of frequency.

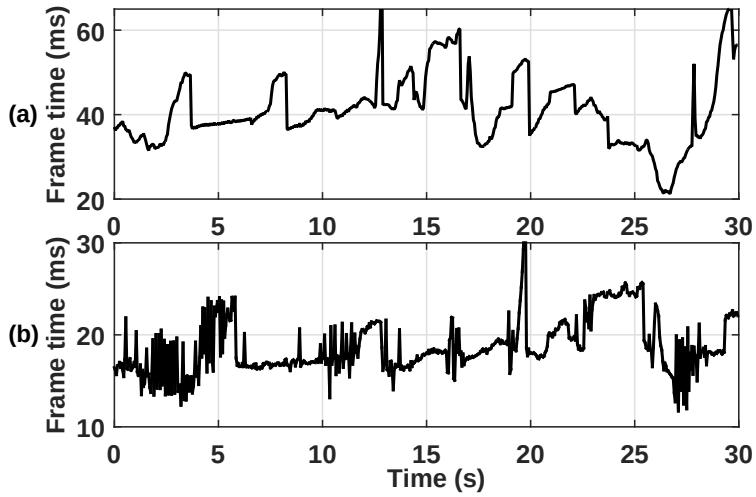

Figure 4.1: The change in frame time for ice-storm application for (a) 200 MHz and (b) 489 MHz GPU frequencies.

In this chapter, we present a performance model that when combined with a power model can be integrated into dynamic power management algorithms to enable selection of the best GPU frequency for graphics applications. We develop a systematic two-step methodology for constructing a tractable runtime model for GPU frame time prediction. The first step is an extensive analysis to collect frame time and GPU performance counter data. This analysis enables us to construct a frame time model template and select the feature set that should be used online. Our model employs differential calculus to express the change in frame time as a function of the partial derivatives of the frame time with respect to the GPU frequency and performance counters. In the second step, we implement an adaptive algorithm, whose function is to learn the coefficients of the proposed model online for dynamically predicting the change in the frame time. Unlike our previous work [40], the proposed adaptive algorithm *does not depend on modeling any performance counters offline*. We achieve this by identifying the counters that depend on the GPU frequency during the offline feature selection process. Hence, we exploit the characterization data, which is already

available, and construct a fully online model without relying on micro-architectural details. Furthermore, we present two different online algorithms that can be employed based on the number of features used in the model. The first algorithm is the covariance form of recursive least squares (RLS) algorithm. RLS is a good choice since the correlation between different frames decays quickly unlike the fractal behavior observed at the macroblock level [142]. The covariance form avoids matrix inversion and incurs very small overhead when the number of features is small ($\approx$10). However, its computational complexity still grows quadratically [85]. Therefore, we also employ the traversal form of RLS with coordinate descent, called Dichotomous Coordinate Descent form of RLS (DCD-RLS), whose complexity grows linearly with the number of features [148]. We employ the adaptive frame time model to estimate the frame time sensitivity to the GPU frequency, which is defined as the partial derivative of the frame time with respect to the GPU frequency.

To validate our approach experimentally, we run custom applications and commonly used graphics benchmarks on three different hardware platforms [1] : the Intel Minnowboard MAX mobile platform [61], Intel core i5 6^{th} generation platform [109], and Moto-X pure edition smartphone [91]. First, we test the accuracy of our performance model. Our experiments show that the mean absolute percentage error in frame time and frame time sensitivity prediction are 4.2% and 6.7%, respectively. Then, we employ our model in a dynamic power management algorithm to optimize energy consumption with performance constraint. We achieve 43% better energy savings than the default Ondemand governor and only 3% higher energy consumption compared to an Oracle policy.

The major contributions of this work are:

[1] Note that our previous work [40] was validated only on the Intel Minnowboard MAX mobile platform.

- A methodology for collecting offline data and developing a GPU performance model,
- An adaptive performance model as a function of the GPU frequency and hardware counters observed online,
- Practical implementation and overhead analysis of two low-cost RLS algorithms to adaptively learn the model coefficients,
- Extensive evaluations of our approach on three experimental and commercial platforms using common GPU benchmarks.

The rest of the chapter is organized as follows. Section 4.2 presents the related work. Section 4.3 details the challenges and lays out the groundwork required for frame time prediction. Section 4.4 presents the techniques for offline analysis and online learning. Finally, Section 4.5 discusses the experimental results, and Section 4.6 concludes the chapter.

4.2 Related Research

The number of power hungry and performance critical graphics applications running on the smartphone is increasing [90]. As a result, power consumption, temperature, and performance metrics in smartphones have become important considerations [38, 99]. Dynamic thermal and power management (DTPM) techniques often perform tradeoffs between these metrics for good user experience [8, 42, 113]. This work focuses on building quantifiable light-weight performance models that can guide DTPM algorithms in conjunction with runtime power models [24, 69, 96].

A number of researchers have proposed dynamic power management techniques for graphics applications [28, 70, 111]. Many of these techniques employ performance

models that are either learned offline or online. For example, Kadjo et al. employs a performance model that is a function of the individual, the products, and the quotients of the hardware performance counters [70]. This technique learns the model parameters using batch linear regression and predicts the frequency-scalable portion of the GPU active time. Thus, enabling accurate performance modeling, but at the same time is dependent on the offline training data. Another work on performance modeling uses an auto-regressive (AR) model for frame time prediction [27]. The authors employ a tenth order AR model, whose coefficients are learned offline using ten minutes of frame time data for each application using the Matlab System Identification tool [84]. In another technique, the authors use a similar AR model whose inputs are based on prior frame times, and the model coefficients are estimated using the normalized least mean squares technique [28].

Workload prediction models based on PID controllers have also shown good accuracy in prediction of graphics workloads [28]. However, as mentioned in [28] the PID gains are very hard to tune due to a large search space of the gain parameters. Furthermore, it is not practically feasible to change the PID gains adaptively at runtime. Yet another approach to compute the GPU performance is presented in [111]. This technique models the GPU performance using the CPU and GPU frequencies and their utilizations as inputs. The authors employ batch linear regression adaptively at runtime to learn the model coefficients, which is computation and memory intensive [123]. Furthermore, their model relies on utilization (instead of using the performance counters) that does not provide a fine-grain measure of the workload.

A hybrid combination of offline and online techniques has recently been proposed to minimize the energy consumption under a performance constraint [88]. This technique employs probabilistic graphical models to estimate the power and performance for unknown applications at runtime based on previously stored offline application

data. The authors show high accuracy compared to an online learning algorithm. However, this online algorithm ignores the application history and employs a basic multi-variable linear regression technique.

In summary, relying solely on offline data does not generalize well to other data sets, as it is not feasible to account for all possible workloads. Alternatively, online learning is challenging due to limited observability and computing resources. We address these concerns by providing an efficient technique for GPU performance prediction, which includes a performance model, a feature selection methodology and an online learning algorithm.

Our adaptive performance model uses hardware performance counters and frequency as inputs. We employ RLS for online learning of the model coefficients. Note that RLS has been extensively applied in signal processing and control applications [123]. In fact, RLS has also been employed for building an adaptive power model [145] and performance model [83, 146] for CPUs. Unlike our work, these models are not built for GPUs, and do not use frequency and performance counters as inputs. Our prior performance model for integrated GPUs [40] also employ RLS algorithm and performance counters. However, it requires offline learning to characterize the frequency dependence of the counters used by the RLS algorithm. More precisely, the prior technique learns a non-linear model offline to compute the derivative of frequency dependent counters with respect to the GPU frequency. Since offline learning limits the usability of the earlier model, we propose a fully online technique. The main challenge is to identify which counters depend on the GPU frequency, and characterize this dependence without knowing micro-architectural details. Our key insight is to find this information in the experimental data set, which is already used for feature selection. We add a subtle term to the model template used in the feature selection step. The new term enables us to choose only the counters that are

not correlated with the frequency term. This leads to a more robust and practical mechanism that employs only frequency dependent counters.

In addition, we present the results with a low complexity DCD-RLS algorithm that can be more efficient than traditional RLS algorithm for large number of inputs [148]. Furthermore, we also evaluate our technique by concurrently running GPU applications on commercial Moto-X pure edition smartphone. Finally, we demonstrate the application of our approach for dynamic power management and evaluate the results on an Intel core i5 6^{th} generation platform.

4.3 Frame Time Characterization

4.3.1 Challenges and Notation

To construct a high fidelity frame time model, it is crucial to understand the dependence of the frame time on the GPU frequency and workload. As mentioned in Section 4.1, the workload characteristics are captured by the performance counters $\mathbf{x} = [x_1, x_2, \ldots, x_N]$, where N is the total number of counters. These counters are functions of the frame complexity C that can be defined as the processing effort required to render a frame. For example, the number of various operations, such as the number of pixels shades, and the number of cycles that the rendering engine is busy vary as the frames stream through the GPU. Consequently, corresponding performance counters become indicators of the frame complexity. Furthermore, both the frame time and some of the counters are functions of the operating frequency. Therefore, we characterize the frame time t_F in any given time step k using a multivariate function $t_{F,k}(f_k, \mathbf{x}_k(C_k, f_k))$, where the subscript k denotes discrete time steps used in practical systems. Besides showing the dependency of the frame time on the frequency and counters, this notation also reveals that the counters themselves can

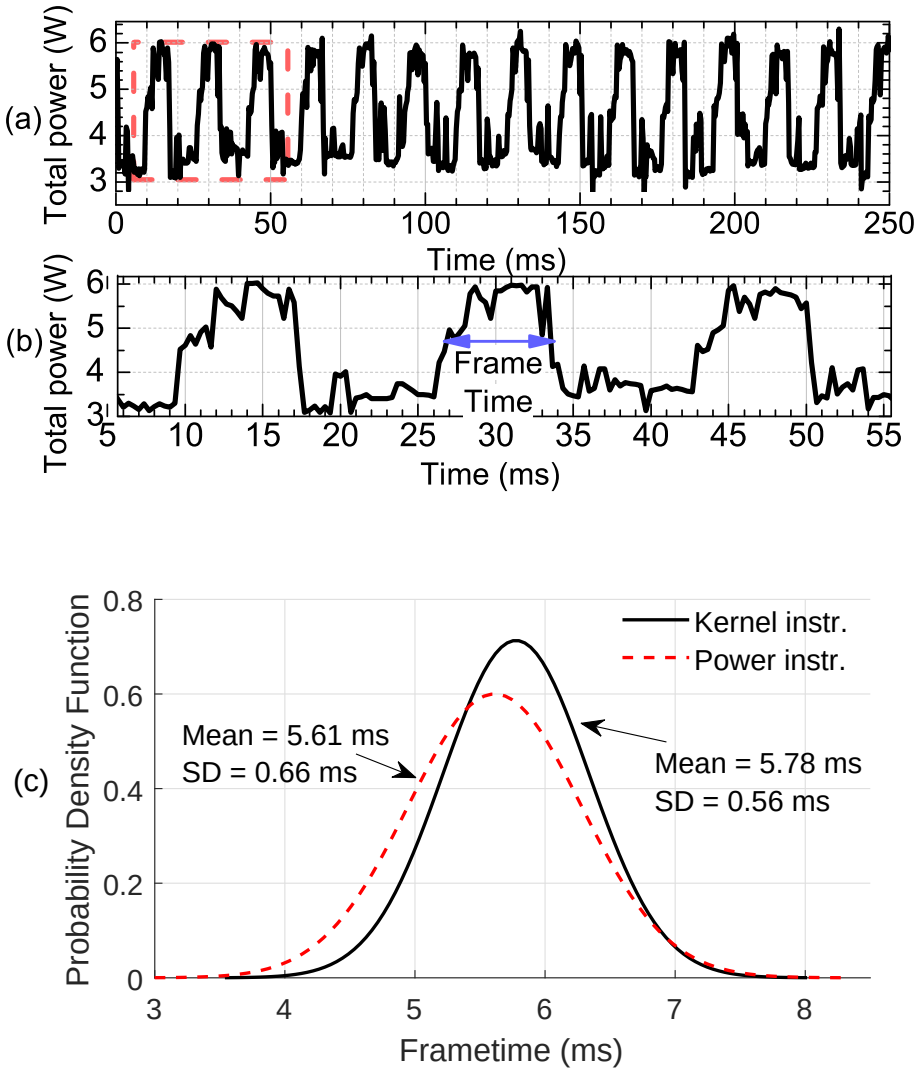

Figure 4.2: (a) Total power consumption of the Intel Minnowboard MAX platform [61] when the GPU is rendering Art3 application at 60 FPS. The crests correspond to the power consumption when the GPU is actively rendering the frames, while the trough correspond to the power consumption when the GPU is in sleep state. (b) Zoomed portion, which shows three frames in the first 50ms. The width of the peaks give the time the GPU is actively computing the frame. (c) Frame time distribution for kernel and power instrumentations for Art3 application.

vary with frequency.

There are two major challenges in the characterization of $t_{F,k}(f_k, \mathbf{x}_k(C_k, f_k))$. The first challenge is to establish a trusted reference for frame time that provides a rich set of samples of this function. This set needs to provide the frame time for an exhaustive list of frequencies and counter values. The second and bigger challenge is to understand the sensitivity of frame time to frequency, *i.e.*, finding the partial derivative of the frame time with respect to the frequency. This quantitative measure of the impact on performance due to a change in the GPU frequency is vital for dynamic power management algorithms. For example, when the derivative is zero, the power management algorithm can safely lower the frequency without affecting the performance. However, finding this partial derivative is very challenging, since a direct reference is not available at runtime. Therefore, we perform extensive offline characterization by decoupling the impact of the change in frame time due to the frequency and frame complexity.

4.3.2 Frame Time and Counter Data Collection

We establish the reference frame time by modifying the Android's Direct Rendering Manager [30] driver to mark the GPU start and completion times for each new frame. In this way, we can record the frame processing time and frame count from the kernel while running any benchmark that uses the GPU. We set the sampling period to 50 ms such that three frames can fit into the interval at 60 frames per second (FPS).

Our frame time instrumentation is a non-trivial modification to the Linux kernel. Therefore, we constructed an experimental setup to validate the accuracy of our instrumentation using power consumption measurements. In our setup, an external power supply is connected to the target platform using a shunt resistor. We employ an

NI data acquisition (DAQ) system [97] to measure the voltage across the terminals of the resistor while running application. Then, the data collected by the DAQ systems is used to compute the current drawn by the target platform. Figure 4.2(a) shows the total platform power consumption as a function of time when running a custom target application (Art3) at 60 FPS. By maintaining a low CPU activity, we know that the peaks in the power consumption occur due to the GPU activity. Figure 4.2(b) zooms to the first 50 ms of the trace that shows three frames. The width of the pulses is a good measure of frame time, since they correspond to the time periods during which the GPU is active. Hence, we can test the accuracy of frame time and frame count instrumentations by correlating them to the pulse durations obtained by power measurements. Figure 4.2(c) shows the probability density functions for the frame time measured by the software kernel instrumentation and the external board power measurements. We observe that our kernel instrumentation and power measurements yield only a 3% difference in mean of the frame time. We also find that the kernel instrumentation is more practical and accurate than the power measurements, since it does not depend on external equipment and has lower measurement noise.

We use the Intel GPU Tools [60] to log the counter values at runtime [59]. Our modification to the kernel source code enables us to collect traces in the format shown below:

Time	Frame Time	Frame Count	GPU Frequency	Perf. Cntr 1	Perf. Cntr 2	...	Perf. Cntr N

Each row corresponds to a 50 ms interval, which matches the rate at which the frequency governors change the GPU frequency. We also test that this data collection does not induce any noticeable impact on the application performance.

Instead of collecting the data every 50 ms interval, another way to isolate the

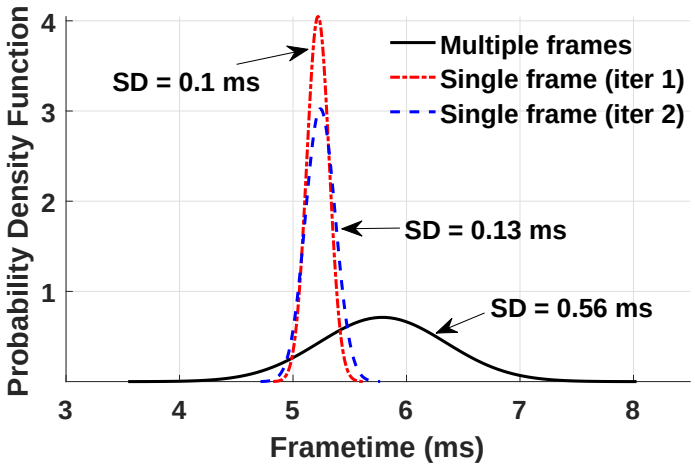

Figure 4.3: The frame time distribution obtained for rendering the same frame and rendering multiple similar frames.

changes due to the GPU frequency is by running the entire application repeatedly at each supported GPU frequency. Theoretically, this data collection method can be used to identify the effect of GPU frequency on frame time. However, this approach is intractable for a number of reasons. First, there will not be a one-to-one correspondence between the frames in different runs. For example, consider an application that runs at 60 FPS or 30 FPS depending on the GPU frequency. At the low GPU frequency, the application will drop the 30 frames that it failed to render, rather than rendering them later. Second, even processing the same frame may take different amounts of time due to the variations in the memory access time from one run to another, as shown in Figure 4.3. We also observe that frame time variations can be significant even when rendering multiple frames that have similar frame complexity. These challenges are aggravated in many GPU intensive applications. Therefore, the most reliable approach for collecting reference data is by varying the GPU frequency while freezing the workload.

A consistent apple-to-apple comparison is possible only if the workload is kept

constant, *i.e.*, same frame is frozen and rendered repeatedly. To facilitate this comparison, we built two custom Android applications, Art3 and RenderingTest, as detailed in Section 4.5.1. These applications enable us to precisely control the frame content and target frame rate. We first set the CPU frequency to ensure repeatability of the results, as shown in Figure 4.4. Then, we sweep the GPU frequency across the set of frequencies supported by the target system. For example, our target platform supports nine frequencies ranging from 200MHz to 511MHz, as shown in Figure 4.4. Each of these combinations is further repeated for 64 frame complexities, which is determined by the number and variety of features in a given frame. We note that different frame complexities enable us to exercise the performance counters in a controlled manner. Finally, we run each configuration multiple times to suppress the random variations. In our experiments, we collect 80 samples for each configuration, which leads to $2 \times 9 \times 64 \times 80 = 92160$ lines with 1152 different configurations.

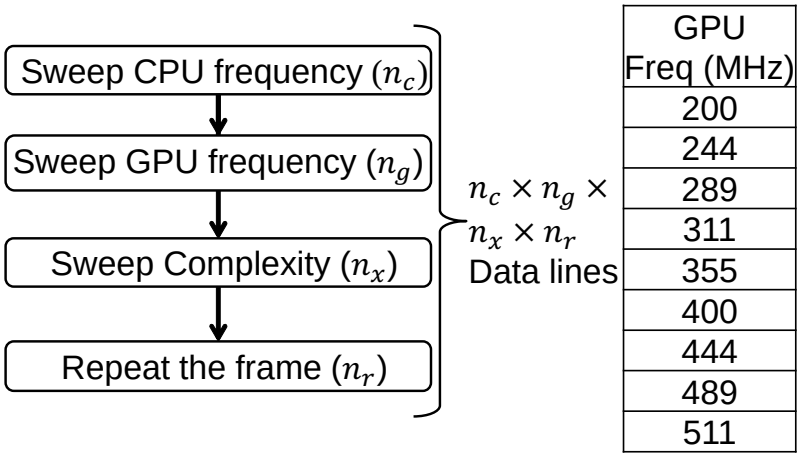

Figure 4.4: The proposed methodology for collecting a rich set of training and test data. Each frame is repeated n_r times for every configuration.

The proposed methodology is applied to both of our Art3 and RenderingTest applications. Figure 4.5(a) shows how the frame time changes with the GPU frequency at a CPU frequency of 1.3 GHz. Different curves on this plot show that increasing

frame complexity implies larger frame time, as expected. Therefore, the data set confirms that the frame time is a function of both the GPU frequency and the workload. Similarly, Figures 4.5(b) and (c) show the *Rendering Engine Busy* counter and *Vertex Shader Active Time* counter as a function of the frequency. The *Rendering Engine Busy* counts the number of cycles for which the rendering engine is not idle and the *Vertex Shader Active Time* counts the cycles for which the vertex shader is active on all cores [59]. Clearly, *Rendering Engine Busy* counter is a strong function of frequency, while *Vertex Shader Active Time* counter is independent of frequency. Figure 4.6 shows the relation between the counters and the frame time. We observe that a larger cycle count (*i.e.,* higher complexity) results in an almost linear increase in frame time. The partial derivative of frame time with respect to the counter value changes with frequency. Furthermore, Figures 4.6(a) and (b) show that the partial derivative of frame time with respect to the counter value, *i.e.*, the slope of the frame time, is a function of both the frequency and counter. In summary, our data set enables characterizing the multivariate function $t_{F,k}(f_k, \mathbf{x}_k(f_k))$. We use this data at design time to construct a template for the frame time model. Then, our online learning algorithm updates the coefficients in this model to predict the frame time for arbitrary applications.

4.4 Frame Time Prediction

This section presents the proposed frame time prediction methodology. First, a mathematical model is derived to express change in frame time, followed by a demonstration of how frame time sensitivity is computed using this model. Then, we describe the offline learning process for selecting the features that will be used during online learning. Finally, we present the proposed adaptive frame time prediction algorithm.

4.4.1 *Differential Frame Time Model*

The quintessential information used by dynamic power management algorithm is: "How do the control parameters (in our case the GPU frequency) affect the performance and power consumption". For example, if the performance is not affected by the GPU frequency, then we can use the minimum available frequency to minimize the power consumption, since there is no performance penalty. In contrast, if the frame time is inversely proportional to the GPU frequency, then it would be prohibitive to reduce the frequency. Therefore, we are interested in modeling the change in frame time as a function of the frequency. From a practical point of view, we know the frame time in the previous interval $k-1$ thanks to our instrumentation. Therefore, the expected change (i.e., the difference from the previous interval) is sufficient to predict the frame time in next interval k. This change can be approximated as the total derivative with respect to the GPU frequency and performance counters as follows:

$$dt_F(f_k, \mathbf{x}_k(C_k, f_k)) = \frac{\partial t_F(f_k, \mathbf{x}_k(C_k, f_k))}{\partial f_k} df_k + \sum_{i=1}^{N} \frac{\partial t_F(f_k, \mathbf{x}_k(C_k, f_k))}{\partial x_i(C_k, f_k)} dx_{i,k}(C_k, f_k) \quad (4.1)$$

This equation reveals that the variation in frame time is a combined effect of the change in the GPU frequency (the first term), and the changes in the counters, which reflect the workload (the summation term). Equation 4.1 holds, if the frequency and counters are continuous variables. Since they are discrete variables in practice, we can approximate the change in frame time as:

$$\Delta t_F(f_k, \mathbf{x}_k(C_k, f_k)) \approx \frac{\partial t_{F,k}}{\partial f_k} \Delta f_k + \sum_{i=1}^{N} \frac{\partial t_{F,k}}{\partial x_{i,k}} \Delta x_{i,k} \quad (4.2)$$

Note that $\partial t_F / \partial f_k$ is the partial derivative of frame time with respect to frequency

[2] . The frame time change due to $\partial x_{i,k}(f_k)/\partial f_k$ is included in the difference term $\Delta x_{i,k}$. This equation forms the basis of our mathematical model. The differential form is useful, since the current frame time is known, and we are interested in the change. Next, we analyze each term of Equation 4.2 in detail to derive our frame time model.

Change due to the GPU frequency: In general, the part of the processing time confined within the GPU pipeline is inversely proportional to the frequency. However, memory access and stall times do not scale with the frequency. Therefore, the frame time is a non-linear function of the GPU frequency, as shown in Figure 4.5(a). Using this observation, we can approximate the frame time t_F *for a given workload* (*i.e.*, $\mathbf{x}$) in terms of a frequency-scalable portion $t_{F,s}$ and an unscalable portion $t_{F,us}$ [7]. More specifically,

$$\begin{aligned} t_F(f_{k-1},\mathbf{x}) &= t_{F,s}(f_{k-1},\mathbf{x}) + t_{F,us}(\mathbf{x}) \\ t_F(f_k,\mathbf{x}) &= t_{F,s}(f_{k-1},\mathbf{x})\frac{f_{k-1}}{f_k} + t_{F,us}(\mathbf{x}) \end{aligned} \tag{4.3}$$

Hence, the change in frame time when switching from f_{k-1} to f_k can be found by subtracting the first line in Equation 4.3 from the second line as follows:

$$\frac{\partial t_{F,k}}{\partial f_k}\Delta f_k \approx t_{F,s}(f_{k-1},\mathbf{x})\left(\frac{f_{k-1}}{f_k}-1\right) \equiv a_0 t_{F,k-1}\left(\frac{f_{k-1}}{f_k}-1\right) \tag{4.4}$$

where $t_{F,k-1}$ is the frame time from the previous instant $k-1$. We note that $t_{F,k-1}\left(\frac{f_{k-1}}{f_k}-1\right)$ can be easily calculated at run time. Since the scalable frame time is in general not known, we express it as *an unknown parameter* a_0 that our online learning algorithm will learn at runtime.

Hardware performance counter change: The frame time changes linearly with many hardware performance counters, such as the one shown in Figure 4.6. If any

[2] We illustrate our approach using a single clock domain, since integrated GPUs used in mobile processors, such as, ARM Mali, have a single clock domain [4]. However, this approach can be extended to multiple clock domains by adding a new frequency term for each clock domain. and using counters representative of all domains.

counters cause a non-linear change in frame time, they can be taken as piece-wise linear. Thus, we express the second term in Equation 4.2, *i.e.,* the change in frame time with counters as:

$$\Delta t_F(\mathbf{x}_k) \approx \sum_{i=1}^{N} \frac{\partial t_{F,k}}{\partial x_{i,k}} \Delta x_{i,k} \equiv \sum_{i=1}^{N} a_i \Delta x_{i,k} \tag{4.5}$$

where a_i's are the coefficients that change at runtime as a function of the workload. Therefore, they are learned online.

By combining Equation 4.4 and Equation 4.5, we can re-write our mathematical model in Equation 4.2 as:

$$\Delta t_{F,k}(f_k, \mathbf{x}_k(f_k)) \approx a_0 t_{F,k-1}\left(\frac{f_{k-1}}{f_k} - 1\right) + \sum_{i=1}^{N} a_i \Delta x_{i,k}(f_k) \tag{4.6}$$

The terms $t_{F,k-1}\left(\frac{f_{k-1}}{f_k} - 1\right)$ and $\Delta x_{i,k}(f_k)\ \forall\ i \in [0, N]$ form the feature set $\mathbf{h}_k$, while the parameters $\mathbf{a} \in \mathbb{R}^{N+1}$ are learned online. The list all of the parameters with their description are shown in Table 4.1.

4.4.2 Frame Time Sensitivity

DTPM algorithms often need to evaluate the impact of a frequency change on performance before making any decision. This information, together with power sensitivity to frequency, can help DTPM algorithms to make better decisions. This section explains how our frame time prediction model is used for computing the frame time sensitivity.

As an example, consider a scenario where the GPU frequency at time k is $f_k = 400$ MHz. Suppose that a DTPM algorithm needs to predict the change in frame time when the frequency goes from $f_k = 400$ MHz to a candidate frequency $f_{\text{new}} = 444$ MHz. Before finalizing this decision, we will need to evaluate the corresponding change in frame time, *i.e.*, $t_F(f_{\text{new}}) - t_F(f_k)$ using Equation 4.6. In this equation,

the frequency change affects the first term $\left(\frac{400}{444} - 1\right)$ and only the counters that are a function of the frequency. To make the latter more explicit, we can write the change in counters due to the GPU frequency f and the frame complexity C as:

$$\Delta x_{i,k} \approx \frac{\partial x_{i,k}}{\partial f}\Delta f_k + \frac{\partial x_{i,k}}{\partial C}\Delta C, \ for \ \ 1 \leq i \leq N \tag{4.7}$$

Since the frame time sensitivity is calculated for a given frame, the change in complexity $\Delta C = 0$, and Equation 4.6 can be written as:

$$t_F(f_{\text{new}}) \ - \ t_F(f_k) \ \approx \ a_0 t_{F,k-1}\left(\frac{f_k}{f_{\text{new}}} - 1\right) \ + \ \sum_{i=1}^{N} a_i \left(\frac{\partial x_{i,k}}{\partial f}(f_{\text{new}} - f_k)\right) \tag{4.8}$$

This equation can be used to predict the change in frame time for the new candidate frequency as:

$$\frac{dt_F}{df}\Big|_k \approx \frac{t_F(f_{\text{new}}) - t_F(f_k)}{f_{\text{new}} - f_k} \tag{4.9}$$

In Equation 4.8, f_k, f_{new}, and $a_i \ \forall \ i \in [0, N]$ are known at time step k. The only unknown value is $\frac{\partial x_{i,k}}{\partial f}$, which is zero for frequency *independent* counters. Note that our prior work employed a non-linear offline model to compute $\frac{\partial x_{i,k}}{\partial f}$ [40]. It is possible to learn this model online as well by employing two parallel adaptive algorithms, but that will incur more overhead. Since it is desirable to keep the overhead of the implementation small, we modify the model to *use only the frequency independent counters*, as described in Section 4.4.3. Selecting the counters for which $\frac{\partial x_{i,k}}{\partial f} = 0$ greatly simplifies the frequency sensitivity calculation. In particular, a simplified form can be obtained after combining the Equations 4.6 and 4.7.

$$\Delta t_{F,k}(f_k, \mathbf{x}_k(f_k)) \approx a_0 t_{F,k-1}\left(\frac{f_{k-1}}{f_k} - 1\right) + \sum_{i=1}^{N} a_i \left(\frac{\partial x_{i,k}}{\partial f}\Delta f_k + \frac{\partial x_{i,k}}{\partial C}\Delta C\right) \tag{4.10}$$

Then, we separate the terms for the change in counters due to frequency f and complexity C.

$$\Delta t_{F,k}(f_k, \mathbf{x}_k(f_k)) \;=\; a_0 t_{F,k-1}\left(\frac{f_{k-1}}{f_k} - 1\right) + \left(\sum_{i=1}^{N} a_i \frac{\partial x_{i,k}}{\partial f}\right) \Delta f_k + \sum_{i=1}^{N} a_i \left(\frac{\partial x_{i,k}}{\partial C} \Delta C\right) \tag{4.11}$$

We combine the term $\sum_{i=1}^{N} a_i \frac{\partial x_{i,k}}{\partial f}$ into a single model coefficient b_1 that is learned at runtime.

$$\Delta t_{F,k}(f_k, \mathbf{x}_k(f_k)) = a_0 t_{F,k-1}\left(\frac{f_{k-1}}{f_k} - 1\right) + b_1 \Delta f_k + \sum_{i=1}^{N} a_i \left(\frac{\partial x_{i,k}}{\partial C} \Delta C\right) \tag{4.12}$$

The change in the counters that are frequency independent can be written as $\Delta x_k = \frac{\partial x_k}{\partial C} \Delta C$. As a result, we can change the summation in the third term to only include the frequency independent counters without loss of generality.

$$\Delta t_{F,k}(f_k, \mathbf{x}_k(f_k)) = a_0 t_{F,k-1}\left(\frac{f_{k-1}}{f_k} - 1\right) + b_1 \Delta f_k + \sum_{i=1}^{N_{\text{indep}}} b_{i+1} \Delta x_{i,k} \tag{4.13}$$

Finally, we perform a change in variables for the model coefficients to represent b with a and obtain the following equation:

$$\Delta t_{F,k}(f_k, \mathbf{x}_k(f_k)) \approx a_0 t_{F,k-1}\left(\frac{f_{k-1}}{f_k} - 1\right) + a_1 \Delta f_k + \sum_{i=1}^{N_{\text{indep}}} a_{i+1} \Delta x_{i,k} \tag{4.14}$$

where $N_{\text{indep}} \subseteq N$ is the number of frequency independent counters. This step simplifies the calculation of $t_F(f_{\text{new}}) - t_F(f_k)$ by making the partial derivative of the counters with respect to frequency equal to zero in Equation 4.14.

$$t_F(f_{\text{new}}) - t_F(f_k) \approx a_0 t_{F,k-1} \left(\frac{f_k}{f_{\text{new}}} - 1\right) + a_1 (f_{\text{new}} - f_k) \tag{4.15}$$

Derivative at time k: We can compute the derivative of frame time with respect to frequency at time k using the average of the derivative to jump one level higher and one level lower frequency. The one level higher and lower frequencies correspond to the smallest possible change in the frequency of the platform.

$$\left.\frac{dt_F}{df}\right|_k = \lim_{\Delta f \to 0} \frac{1}{2}\left[\frac{t_F(f_k + \Delta f) - t_F(f_k)}{\Delta f} + \frac{t_F(f_k) - t_F(f_k - \Delta f)}{\Delta f}\right] \tag{4.16}$$

where Δf is the change in the frequency one level higher and lower to the frequency f_k. Since the change in the frequency is in both the higher and lower directions, the weights are 0.5. For some platforms, such as Minnowboard the frequency levels are not equally spaced. For example, when $f_k = 489$ MHz the change to the frequency one level higher is $\Delta f_1 = 511 - 489 = 22$ MHz and one level lower is $\Delta f_2 = 489 - 444 = 45$ MHz, as shown in the frequency table of Figure 4.4. To accurately predict the numerical derivative of frame time with respect to the frequency, we employ a three point derivative of Lagrange's polynomial [127, 135] as follows:

$$\left.\frac{dt_F}{df}\right|_k \approx \frac{\Delta f_1^2 t_F(f_k + \Delta f_2) + (\Delta f_2^2 - \Delta f_1^2) t_F(f_k) - \Delta f_2^2 t_F(f_k - \Delta f_1)}{\Delta f_1 \Delta f_2 (\Delta f_1 + \Delta f_2)} \tag{4.17}$$

Equation 4.17 simplifies to Equation 4.16 for equal spacing of frequencies, *i.e.*, when $\Delta f_1 = \Delta f_2$.

4.4.3 Offline Feature Selection

Real-time prediction requires an extremely efficient learning algorithm to facilitate fast evaluation of a GPU frequency change. One approach to reduce the overhead of regression is dimensionality reduction on the input data. The goal of this approach

is to reduce the complexity of the data and speed up computation, while maintaining a good prediction accuracy. In addition to algorithm efficiency, this can help remove the features that either add duplicate information to the output or do not change with our parameters. The main challenge here is to *identify which counters depend on the GPU frequency* and *characterize this dependence without knowing micro-architectural details*. We note the Equation 4.14 has two types of terms. The first two terms with coefficients a_0 and a_1 are explicit functions of the frequency, whereas the remaining terms are functions of the performance counters. If the counters in our feature set are correlated with the frequency, RLS cannot reliably converge to optimal model coefficients due to the multicollinearity phenomenon. Therefore, we limit our feature set to the performance counters that are independent from the frequency. We are able to differentiate frequency dependent and independent counters using our characterization data without having access to the micro-architecture of the GPU. We employ Least Absolute Shrinkage and Selection Operator regression (Lasso) to reduce the feature size in the model appropriately by selecting the most representative set of features by minimizing the MSE with a bound on the ℓ_1 norm of parameters a_i [34]. The results from Lasso regression are highly sparse due to the ℓ_1 nature of the bound. That is, for T samples the Lasso regression can be performed by minimizing the MSE between the actual change in frame time $\Delta t_{F,k}$ and using the estimate from Equation 4.14 after adding a ℓ_1 norm penalty as:

$$\hat{a} = \underset{a}{\operatorname{argmin}} \sum_{k=1}^{T} \left[\Delta t_{F,k} - a_0 t_{F,k-1} \left(\frac{f_{k-1}}{f_k} - 1 \right) - a_1 \Delta f_k + \sum_{i=1}^{N_{\text{indep}}} a_{i+1} \Delta x_{i,k} \right]^2 + \eta \sum_{j=0}^{N_{\text{indep}}} |a_j| \quad (4.18)$$

By increasing the value of η, less features can be selected at the expense of accuracy. An acceptable loss in accuracy is within one standard error more than the minimum MSE. Thus, during the learning phase we will regress on M feature sub-

set, where $M << N + 1$, instead of $N + 1$ features. Note that our approach relies on the availability of frequency independent features in the platform. Based on our experiments with Minnowboard [61] and Intel core i5 6^{th} generation platform [109], we have always been able to find frequency independent features.

4.4.4 Online Learning of the Model Parameters

The parameters in Equation 4.14 can be learned offline and then used at runtime. However, it is hard to generalize offline learning to all possible applications that would be executed by the system. Moreover, the workload can change as a function of user activity. Therefore, the learning mechanism should not completely rely on offline learning. We employ an adaptive algorithm to learn the parameters of the frame time model. In particular, we use the covariance form of RLS [123] and the Dichotomous Coordinate Descent form of RLS [148] estimation techniques, as described next.

RLS algorithm updates the parameters a_i in Equation 4.14 in each prediction interval, as described in Figure 4.7, using the following set of equations:

$$\mathbf{G}_k = \mathbf{P}_{k-1}\mathbf{h}_k(\mathbf{h}_k^T\mathbf{P}_{k-1}\mathbf{h}_k + \lambda)^{-1} \tag{4.19}$$

$$\mathbf{P}_k = (\mathbf{I} - \mathbf{G}_k\mathbf{h}_k^T)\mathbf{P}_{k-1}\lambda^{-1} \tag{4.20}$$

$$\hat{\mathbf{a}}_k = \hat{\mathbf{a}}_{k-1} + \mathbf{G}_k(\Delta t_{F,k}(f_k, \mathbf{x}_k(f_k)) - \mathbf{h}_k^T\hat{\mathbf{a}}_{k-1}) \tag{4.21}$$

The update rule given in Equation 4.21 computes the prediction error by subtracting the frame time prediction from the actual change in frame time. Note that online learning would not be possible without our kernel instrumentation, which provides *reliable reference measurement at runtime* ($\Delta t_{F,k}(f_k, \mathbf{x}_k(f_k))$). Equation 4.19 and Equation 4.20 update the gain $\mathbf{G}_k$ and covariance $\mathbf{P}_k$ matrices using the feature vector. The forgetting factor $0 \ll \lambda \leq 1$ is used to give more weight to latest data

and less weight to the older data. The set of Equations 4.19-4.21 together solve the ℓ_2 regularized cost function at runtime for any samples T as follows [67]:

$$J = \min_{\mathbf{a}} \left[(\mathbf{a} - \mathbf{a}_{\text{init}})'(\mu \mathbf{I})(\mathbf{a} - \mathbf{a}_{\text{init}}) + \sum_{k=1}^{T} (\Delta t_{F,k} - \mathbf{h}_k' \mathbf{a})^2 \right] \tag{4.22}$$

where $\mathbf{a}_{\text{init}}$ are the initial values of the model coefficients $\mathbf{a}$ and μ is a regularization parameter. We denote the matrix and vector transpose by $(\cdot)'$ symbol.

Parameter initializations: We choose the $\mathbf{a}_{\text{init}} = diag(\mathbf{I})$, since we assume full scalability of the frames with respect to the frequency and counters in the beginning. The forgetting factor λ is set to one to utilize all the past information. We find the regularization parameter μ such that the multicollinearity of the inputs is considerably reduced. Multicollinearity in linear regression problems occur when two or more inputs are highly correlated causing the standard errors in the estimate of the coefficients to increase [31]. RLS solves the multicollinearity issue by minimizing a ℓ_2 regularized cost function [55, 67]. Finally, we initialize the covariance matrix as $\mathbf{P} = \mathbf{I}/\mu$.

Computational complexity: RLS is well known for giving good predictions in the signal processing field. However, its computational complexity grows with the number of features as $O(M^2)$ [124]. Nonetheless, feature selection minimizes the size of the feature set to reduce the complexity. Furthermore, matrix inversions are the main source of complexity in many algorithms, including RLS. Our solution is to use the co-variance form of RLS that does not perform matrix inversion. The value $\mathbf{h}_k^T \mathbf{P}_{k-1} \mathbf{h}_k$ in Equation 4.19 evaluates to a scalar, eliminating the overhead of the inversion operation. The complexity of the RLS is acceptable for small number of features. When there are large number of features then a traversal form of RLS coupled with coordinate descent called DCD-RLS can be used [148]. In this algorithm, first, the correlation matrix $\mathbf{P}^{-1}$ is partially updated in each time stamp k. Then, the

change in the model coefficients are estimated using inexact line-search. This reduces the complexity of the DCD-RLS algorithm to O(M). For example, in a platform if the number of features $M = 10$, then the number of arithmetic operations in RLS are $2M^2 + 8M + 2 = 282$, while the operations used in DCD-RLS are only $17M = 170$. Since in our current platform we perform feature selection and reduce the number of features to 4, the number of operations in RLS and DCD-RLS are similar. Also, DCD-RLS reduces the number of multiplication and division operations at the expense of low-cost addition operations. This provides slight speedup for small features and larger benefits when the number of features are more. More details about the platform overhead of RLS are given in Section 4.5.7.

4.5 Experimental Results

This section first describes the experimental setup and the selection of the offline learning of regularization parameters η and μ. Next, we demonstrate the accuracy of the proposed online frame time and frequency sensitivity prediction techniques. After that, we compare our approach to an existing online performance modeling methodology, and demonstrate its application for dynamic power management. Finally, we discuss the implementation overhead of the proposed frame time prediction techniques.

4.5.1 Experimental Setup

We primarily employ the Minnowboard MAX platform [61] running the Android 5.1 operating system with the kernel modifications mentioned in Section 4.3.2 to evaluate our approach. This platform has two CPU cores and one GPU, whose frequency can take the values listed in Figure 4.4. The GPU frequency is readily available from the kernel file system. In addition to this, we use the Intel GPU Tools

as an external module to the Android system to trace the GPU performance counters. To further demonstrate the effectiveness of our approach, we employ two additional hardware platforms. We evaluate the accuracy of our approach while running multiple graphics applications concurrently using a Moto-X pure edition smartphone, which has Qualcomm Snapdragon 808 SoC. Finally, we employ Intel core i5 6^{th} generation platform [109] for dynamic power management experiments.

Standard Benchmarks and Scenarios: The proposed frame time prediction technique is validated using the following commonly used GPU benchmarks on Minnowboard MAX platform: Nenamark2, BrainItOn, 3DMark (both the Ice Storm and Slingshot scenarios), Mobilebench, Chess, and Jet-ski. We also employ eight gaming application scenarios, such as Fruit Ninja, Angry Birds, Jungle Run, Angry Bots, and Shark Dash, running on Intel core i5 6^{th} generation platform. These workloads are referred to as Workloads 1-8 for confidentiality [3] . Finally, we run YouTube application and Chain Reaction game concurrently using Android 7 split-screen feature to create a multiple application scenario on Moto-X pure edition smartphone.

Custom Benchmarks: The accuracy of the frame time prediction can be tested without any limitations, since our frame time prediction technique works for any Android app that can run on the target platform. However, validating the sensitivity prediction (*i.e.,* the partial derivative of the frame time with respect to the frequency) requires reference measurements taken at different frequencies. This golden reference cannot be simply collected by running the whole application at different frequencies due to the reasons detailed in Section 4.3.2. Therefore, we also developed RenderingTest and Art3 applications that enable us to control the number of times each frame is repeated.

The RenderingTest application accepts two inputs that specify the number of

[3]This is requested by Intel Corp.

cubes rendered in the frame, and the number of times the same frame is processed. By changing the number of cubes, we control the frame complexity. In our experiments, we sweep the number of cubes from 1 to 64 and repeat each frame 80 times. The cubes are rendered at a maximum of 60 FPS with vertex shaders and depth buffering enabled. Since we use the RenderingTest application for offline characterization, we also developed one more custom application, called Art3, which renders pyramids with a different rendering pipeline. The RenderingTest application renders each cube with its own memory buffer, while Art3 concatenates all pyramids into the same memory buffer before rendering. These two applications allow us to compute and store the reference sensitivities, such that they can be used as the golden reference to validate our online frequency sensitivity predictions.

Evaluation: We evaluate the proposed methodology using three algorithms. The first algorithm employs Equation 4.14 with online learning using RLS algorithm (RLS). This is also the default algorithm used throughout the chapter. The second algorithm employs the same model with online learning using the DCD-RLS algorithm (DCD-RLS). The third algorithm employs two models: (a) the model shown in Equation 4.6 with online learning using RLS and (b) an offline nonlinear model for derivative of frequency dependent counter with respect to frequency (RLS+Offline) [40].

4.5.2 Offline Feature Selection and ℓ_2 Regularization

To perform feature selection using Equation 4.18, we first prune the counters that are highly dependent on frequency by measuring the Pearson correlation coefficient of the counters with respect to the GPU frequency. Counters that have the correlation coefficient less than 0.1 are retained for further processing. Then, we apply the Lasso regression with 10–fold cross-validation on our large dataset collected from the

RenderingTest application. Figure 4.8(a) shows the change in mean squared error between the predicted and measured frame time of the GPU. As the ℓ_1 regularization parameter η in Equation 4.18 increases, the penalty on the cost function increases leading to a higher MSE, in general. Note that the mean error (black line) first slightly decreases, then increases for incrementing η values. The slight decrease occurs due to overfitting that also leads to higher cross-validation variance in the error. The minimum value of $\eta_{\text{min}} = 5 \times 10^{-3}$ uses four features, as shown in Figure 4.8(b). To shrink the model features, a good choice is $\eta_{\text{sel}} = 3.4 \times 10^{-3}$ for which the performance in terms of expected generalization error is within one standard error of the minimum. In our experiments, we choose the minimum MSE point with four features. These four features consist of the two change in the frequency terms from Equation 4.14 and change in the *Vertex Shader Active Time* and *Slow Z Test Pixels Failing* counters. The Vertex Shader Active Time counter counts the cycles for which the vertex shader is active on all cores. The Slow Z Test Pixels Failing counter gives the number of pixels that fail the slow check in the GPU. Neither of these counters depend on the frequency; they are functions of only the frame complexity. Note that in our prior work [40] we also select four features, but these consist of a single frequency change term and three counters. Two of these counters, *Aggregate Core Array Active* and *Slow Z Test Pixels Failing* are frequency independent and one counter *Rendering Engine Busy* is frequency dependent. We compute the derivative of the frequency dependent counter offline using a non-linear model. However, in this work by using frequency independent counters only, there is no need for using any additional models. Figure 4.9 shows the features employed by our GPU performance model. We observe that all the features are highly correlated to the change in frame time.

We determine the ℓ_2 regularization parameter μ for optimizing the cost function in Equation 4.22 of the RLS algorithm offline. We first sweep the parameter μ between

a large range of 10^{-28} to 10^{20}, and run the RLS algorithm for each value of the μ to find the error in the frame time predictions. Figure 4.10 shows the mean and variance of the absolute percentage error in frame time for a number of μ values for the RenderingTest and Art3 applications. When μ is small, there is little regularization effect and consequently the error is low. However, when μ value is large, the left term in Equation 4.22 starts to dominate the cost function and severely constrains the model coefficients $\mathbf{a}$ close to $\mathbf{a}_{\text{init}}$. This leads to higher frame time prediction errors for $\mu > 1$. We employ a $\mu = 10^{-14}$ in all our experiments, which is the geometric mean of the starting sweep value of $\mu = 10^{-28}$ and the knee point $\mu = 1$ to provide sufficient adaptability for any unknown workloads.

4.5.3 Online Frame Time Prediction

We validate our frame time prediction approach first on the RenderingTest application to test the corner cases. Figure 4.11 shows the comparison between the actual and the predicted frame time. During the first 5 seconds, both the GPU frequency and frames change randomly. We observe that the proposed online model successfully keeps up with the rapid changes. In order to test our approach under corner cases, we enforce a saw-tooth pattern during the remaining duration of the application. More precisely, the GPU frequency starts at 200 MHz, and the complexity increases from 1 to 64 in increments of one (the first tooth). Then, the same iterations are repeated for 9 supported GPU frequencies. Figure 4.11 demonstrates that we achieve very good accuracy when the frequency stays constant for a period of time. There is a spike when the complexity jumps suddenly from 64 to 1. However, the RLS reacts quickly and maintains a high accuracy. Overall, the mean absolute percentage error between the real and predicted frame time values is 2.6%.

We observe similar levels of accuracy for Art3 and standard benchmarks. In

particular, Figure 4.12 shows the actual and predicted frame times for 3DMark's Ice Storm benchmark at two different GPU frequencies. We achieve a high prediction accuracy with the mean absolute error of 2.1% and 7.4% for the GPU frequencies 200 MHz and 489 MHz, respectively. Similarly, the actual and predicted frame time for the BrainItOn gaming application with fixed GPU frequency is shown in Figure 4.13. This interactive game requires frequent user inputs, and the frame time exhibits more sudden changes compared to other applications. Our frame time prediction matches closely to the actual frame time with the median and mean absolute percentage errors of 0.4% and 12.9%, respectively. Note that the higher mean absolute error value for the application is due to a few outliers in the frame time. This is confirmed from the very low median absolute percentage error value of the benchmark.

The frame time prediction mean absolute error for all of the benchmarks running over *all GPU frequencies* is summarized in Figure 4.14. The results are sorted with the errors in the RLS technique. The average of the mean absolute errors across all the benchmarks for the RLS, RLS+Offline, and DCD-RLS algorithms are 4.2%, 4.3%, and 4.6%, respectively. On average, the three algorithms provide similar and high accuracy. The RLS and DCD-RLS techniques have the additional advantage of not relying on any offline model. We observe that the games BrainItOn and Jetski require extensive user interaction, which leads to fast changes in the frame time. This makes the tracking of the rapidly changing frame time difficult and results in a mean error of 12% and 10%, respectively. Nonetheless, both these applications have low median absolute errors of 6.5% and 1.3%, which suggests that the error is not high for majority of time intervals. Other benchmarks show errors smaller than 5%, indicating very high accuracy for frame time prediction. For Scenario-4 benchmark the DCD-RLS technique shows 3% higher error compared to the other two algorithms. This is because the RLS algorithm is better at rejecting the noise in

the inputs compared to the DCD-RLS. This indicates that RLS should be preferred over DCD-RLS, except when the complexity of RLS is critically important in the system and slightly larger errors in frame time prediction are acceptable.

Comparison with Completely Offline Learning: We also compare our approach with an offline method, where all the model parameters are learned at design time and remain constant at runtime. Figure 4.15 shows the mean absolute percentage errors for online (dashed line) and offline (solid line) learning for different training ratios. When we run all the benchmarks one after the other with our online learning mechanism, we get an error of 4.6%. However, running the same benchmarks with offline learned parameters leads to higher errors. As shown in the figure, the difference between the offline and online error decreases as the training ratio approaches one, *i.e.*, when the training set equals the test set. This shows that offline learning leads to higher error, unless the model can be trained on all the applications. Of note, the prediction error of our approach is flat, since the same set of features are selected with smaller training set.

Frame Time Prediction for Concurrent Application: Newer generation of mobile platforms using Android 7.1 have added support for running multiple applications using split-screen. Therefore, it is important to also validate the performance model on these newer generation of platforms and multiple application scenarios. For this experiment, we employ the Moto-X pure edition smartphone running Android 7.1 on a Qualcomm Snapdragon 808 SoC. We split the screen into two parts as top and bottom. Then, we run a YouTube application in the bottom part of the screen and play the Chain reaction game simultaneously on the top part of the screen. Figure 4.16 shows the reference and predicted frame times for this multiple application scenario. The proposed RLS algorithm achieves 8% frame time prediction error.

There are many benefits of the online performance model compared to offline

evaluation. For example, in our case, the online modeling methodology reduced the characterization and model tuning effort from several months to a few days for the Moto-X smartphone. Similarly, the mobile platforms are expected to deliver good performance for any new applications that were created after the product launch. Therefore, the online modeling technique enables adaptation to the new workloads without costly repetition of the workload characterization by the platform designers. Finally, our approach is easily portable and independent of any vendor and architecture.

4.5.4 Online Frame Time Sensitivity Prediction

To assess the accuracy of our sensitivity prediction, we predict the change in frame time as a result of increasing (or decreasing) the frequency. Then, we compute the frame time sensitivity using Equation 4.9. We start with changing the frequency by one level according the supported GPU frequencies listed in Figure 4.4, *e.g.,* changing f_{GPU} from $f_k = 400$ MHz to $f_{new} = 444$ MHz or $f_{new} = 355$ MHz. Figure 4.17 shows the predicted and actual frame time when the new frequency f_{new} is one level higher. The mean absolute percentage error for this prediction is 5.4%. We observe the same result when f_{new} is one level lower. One might argue that the high prediction accuracy is only due to single frequency jumps like 400 MHz to 444 MHz. Therefore, we also repeat our experiments for multiple frequency jumps. For example, if current frequency is 200 MHz, then a frequency jump of three implies f_{new} is 311 MHz. Figure 4.18 shows that the accuracy indeed degrades, but even when the number of frequency levels is eight (maximum allowed on Minnowboard), the error is less than 10%. In practice, the frequency level changes in DTPM algorithms is not performed drastically from lowest to highest, but in smaller steps leading to higher accuracy.

Accuracy of the Partial Derivative of Frame Time with Respect to Fre-

quency: We present the accuracy in predicting the derivative of frame time with respect to GPU frequency for the RenderingTest and Art3 applications in Figure 4.19(a) and (b), respectively. Each plot shows the derivative values for the reference, RLS, RLS+Offline, and DCD-RLS techniques. We compute the derivative using Lagrange's polynomial method with change in frequency one level higher and one level lower, as given by Equation 4.17. As seen from Figure 4.19(a), the slope starts with a negative value and then diminishes to zero on increasing frequency. This is consistent with the observation in Figure 4.5(a). The normalized root mean squared error in the derivative prediction for RenderingTest application using RLS, RLS+Offline, and DCD-RLS are 6.8%, 6.9%, and 5.9%, respectively. These results indicate high accuracy for the derivative prediction, with the RLS and DCD-RLS having an additional advantage of performing the prediction completely online without using frequency dependent counters. This eliminates an extra step of predicting the derivative of the counter with respect to frequency. In addition to running the RenderingTest application, we ran Art3 as well to measure frame time sensitivity. Figure 4.19(b) shows that the predicted derivative of frame time with respect to GPU frequency follows the reference values closely. In particular, the normalized root mean squared error in the derivative prediction for Art3 application using RLS, RLS+Offline, and DCD-RLS algorithms are 6.6%, 4.9%, and 8%, respectively. Off note, the derivative values for Art3 application are smaller than the RenderingTest application, because Art3 is a memory bound graphics application.

4.5.5 Comparison with an Auto Regressive Model using LMS

In this section, we compare our approach to a tenth-order autoregressive (AR) model which learns the model parameters using Normalized Least Mean Square (LMS) algorithm [28]. We first observe that LMS algorithm is slower to converge

than RLS. For example, Figure 4.20 shows that our approach converges to optimal model coefficients in $50ms$ while running the Icestorm application. In contrast, the LMS approach takes $1.6s$ to converge while running the same workload. In general, the optimal model coefficient targets also change at runtime as the application phases change dynamically. The convergence of the LMS approach is slow due to the tenth-order AR model, which takes the first ten samples to do the initial learning. However, the convergence time of our approach varies between $50ms$ to $0.3s$, while LMS takes in the order of seconds. We also note that the AR model can predict the frame time, but it cannot predict the partial derivative of GPU performance with respect to frequency, since it does not have a frequency term. Therefore, our approach can directly provide the frequency sensitivity data to dynamic power management algorithms unlike the existing AR model [28]. Furthermore, fast convergence enables quick response to the dynamic changes in the workload.

4.5.6 Impact for Dynamic Power Management

Our performance model can be used with a large variety of power management algorithms that can optimize for system objectives, such as performance under a power budget [47, 150] and energy [46]. In this section, we demonstrate the application of the proposed GPU performance model for minimizing the energy consumption subject to a minimum frame rate constraint of 60 FPS. At each control interval, we use the proposed GPU performance model to predict the frame time at all the frequencies supported in the platform. Then, we select the frequency that leads to the smallest energy consumption, while meeting the minimum frame time constraint for the next interval. To evaluate the effectiveness of our approach, we compare our results to an Oracle-based policy that precisely knows what the frequency in the next interval should be. We obtain this information by running each frame at each

supported frequency before this experiment. Obviously, Oracle-based policy is not practical, but it provides the optimal results as a comparison point. In addition to Oracle, we also compare our approach against the Linux Ondemand governor, which is used in many commercial products [106].

For this experiment, we run industrial gaming workloads and our custom applications [4] . Out of these interactive games, the first five workloads have frame time error less than 4%, while the remaining workloads 6-8 have higher frame time errors of more than 10%. Figure 4.21 shows the energy consumption achieved by the Ondemand governor and the proposed RLS-based algorithm. The optimal energy value achieved by Oracle is shown by the dotted red line, and the other results are normalized to that of the Oracle-based policy.

Workloads-6 to 8 and our custom applications have light-load graphics processing requirements. Consequently, these applications have low GPU utilization and can achieve the 60 FPS frame rate target with small GPU frequencies. Our algorithm successfully chooses the right GPU frequency and matches the Oracle-based policy, as expected. The Ondemand governor, which makes its decisions based on the GPU utilization, chooses small frequencies. As a result, it can also achieve the minimum energy consumption.

Unlike the light-load graphics applications, the frame rate target cannot be achieved with lower GPU frequencies while running Workloads-1 to 5. These workloads are heavy to medium-load graphics games that result in high GPU utilization. In this case, high GPU utilization makes the Ondemand governor choose large frequencies. As a result, its energy consumption is 1.3×-2.6× larger than the minimum energy achieved by the Oracle-based policy. In contrast, our RLS-based approach can suc-

[4] These workloads include games, such as, Fruit Ninja, Angry Birds, Jungle Run, Angry Bots, and Shark Dash, running on Intel core i5 6^{th} generation platform. We refer to these games as Workload 1-8 in the plot for confidentiality following the request from Intel.

cessfully choose the optimal operating frequencies due to its high accuracy. Consequently, the energy consumption of our approach is within 1.06× of the optimal value.

Overall, our RLS-based policy leads to only 3% higher average energy consumption compared to the Oracle-based policy. In contrast, the Ondemand governor has 1.3×-2.6× larger energy consumption under heavy workloads. On average, our RLS-based policy provides about 43% lower energy consumption compared to the Ondemand governor while achieving the same frame rate.

4.5.7 Overhead Analysis

We measure the overhead of the proposed approach by instrumenting the start and end times of each of the RLS iterations, including the feature data preparation step. Then, we measure the time for the proposed frame time prediction mechanism running on the Minnowboard platform. Figure 4.22 demonstrates the difference in the runtime overheads of the RLS and DCD-RLS algorithms in each iteration. When the number of features are four, the overhead time of the RLS and DCD-RLS algorithms are 3.8μs and 3.2μs, respectively. As the number of features increase to 20, the runtime overhead of the RLS algorithm becomes much larger than DCD-RLS. More precisely, for 20 features, the RLS algorithm has the runtime overhead of 53.4μs, while the DCD-RLS algorithm has 7.6× smaller overhead of 7μs. This experiment demonstrates that the proposed RLS technique has very low overhead for a small number of features. When the number of features are large and lowering the overhead time is critical, DCD-RLS is a viable alternative to the proposed RLS algorithm.

4.6 Conclusion

In this chapter, we propose an online performance modeling methodology for graphics cores. The proposed methodology combines offline data collection and online learning using RLS algorithm. Online learning of the model coefficients enables adapting to unknown workloads by eliminating the need for costly offline training. Extensive evaluations on an experimental platform using common GPU benchmarks resulted in average mean absolute errors of 4.2% in frame time and 6.7% in frame time sensitivity prediction. Furthermore, we experimentally showed that the proposed high accuracy performance model could be successfully employed by an dynamic power management algorithm that minimizes energy consumption under a performance constraint.

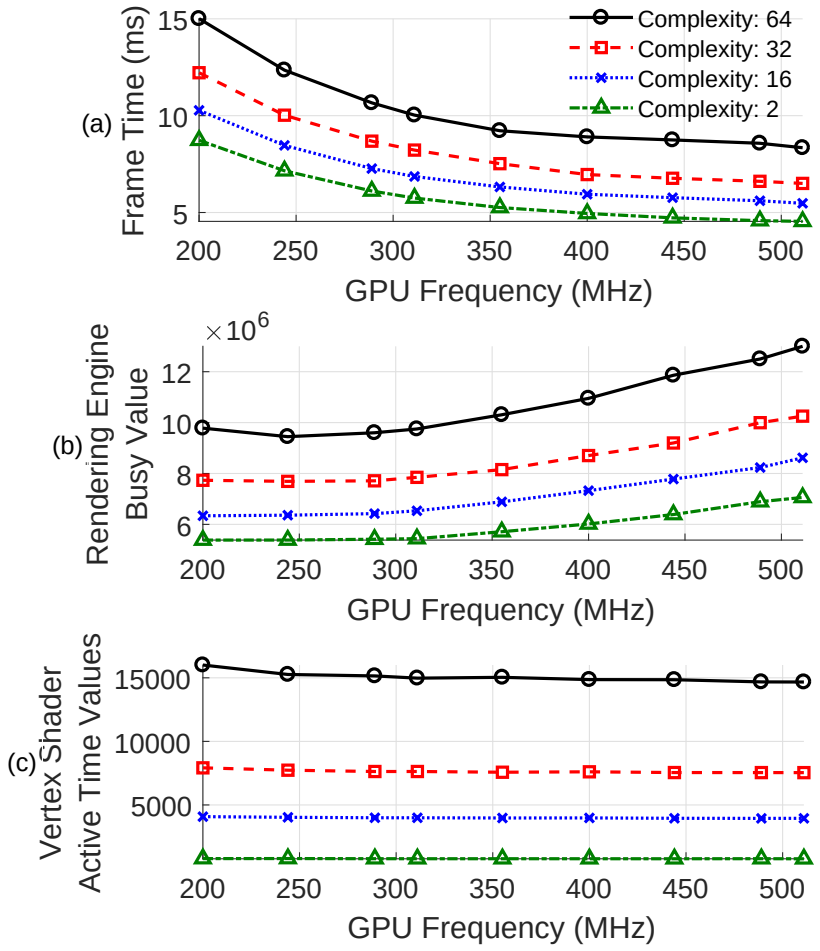

Figure 4.5: Frame time and hardware counter values for the RenderingTest application with increasing GPU frequency at four different frame complexities.

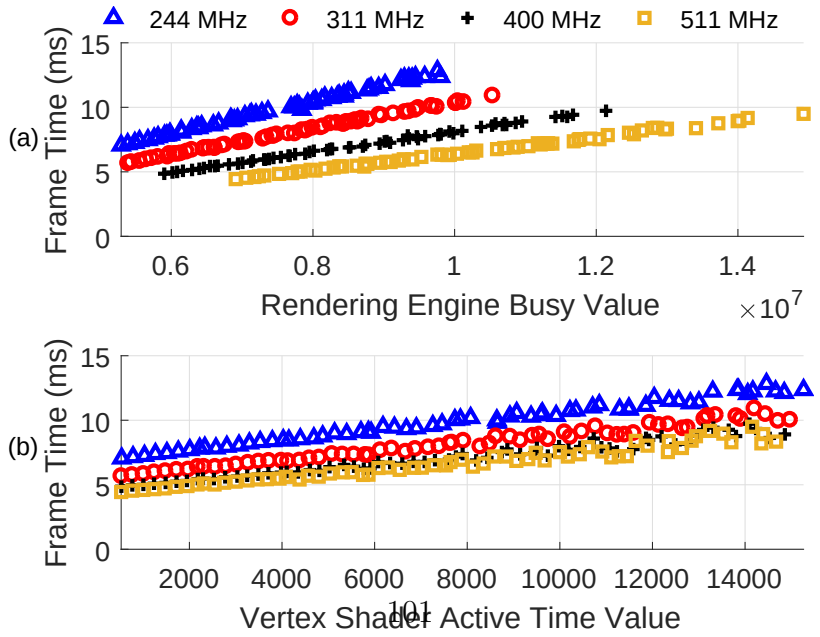

Figure 4.6: Frame time for the RenderingTest application with increasing frame complexity at four different GPU frequencies.

Table 4.1: Summary of the notation used in this chapter

Notation	Description
k	Discrete time sample
f	GPU frequency
C	Complexity of a frame
$\mathbf{x} = [x_1, \ldots, x_N]$	A vector of N hardware counters
T	Total number of data samples
t_F	Frame time
$t_{F,s}$	Frequency-scalable portion of frame time
$t_{F,us}$	Unscalable portion of frame time
$\mathbf{a}$	Model parameters
f_{new}	A new candidate GPU frequency
N_{indep}	Number of frequency independent counters
η	ℓ_1 regularization parameter
M	Number of features after offline selection
$\mathbf{G}$	Adaptive gain of the RLS
$\mathbf{P}$	Covariance matrix of the error in RLS
$\mathbf{h}$	Input features
λ	Forgetting factor
$\mathbf{a}_{\text{init}}$	Initial estimate of the model parameters
μ	ℓ_2 regularization parameter for RLS

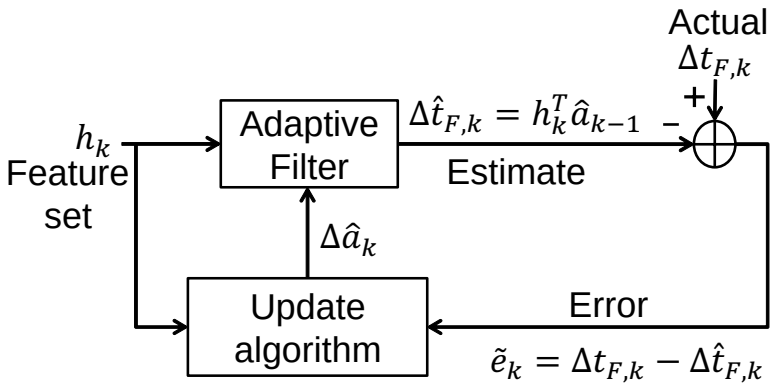

Figure 4.7: Adaptive filtering approach showing the update in parameters a_i based on error between the actual change in frame time and prediction.

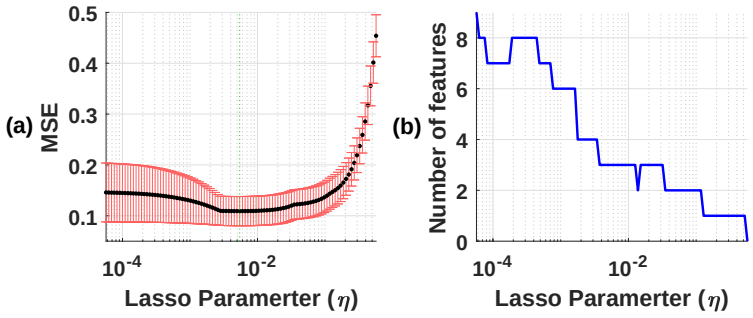

Figure 4.8: Cross-validated LASSO regression result for; (a) the change in mean squared error of the frame time prediction with increasing η values, and (b) the change in the number of selected features with increasing η values.

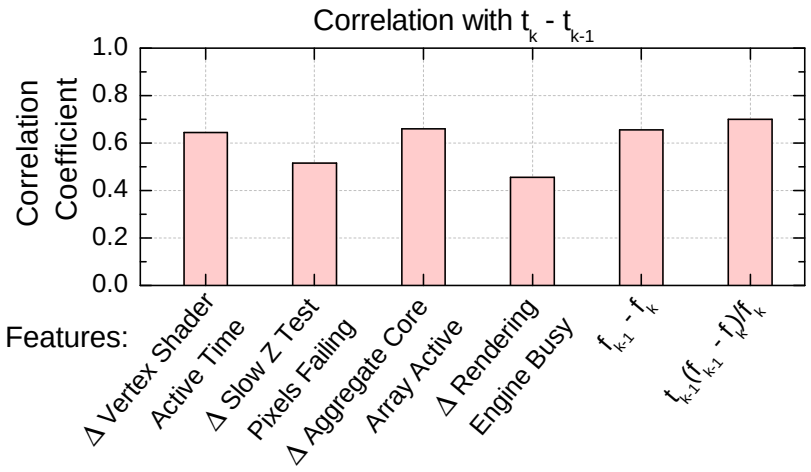

Figure 4.9: Correlation between the selected features and the difference in the frame time $t_k - t_{k-1}$.

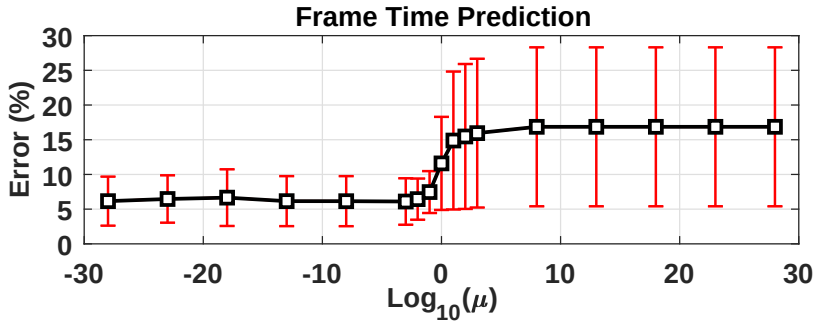

Figure 4.10: Frame time prediction error for RenderingTest and Art3 applications for different values of the ℓ_2 regularization parameter μ. The black markers show the mean value of the error and the whiskers show the one standard deviation boundaries.

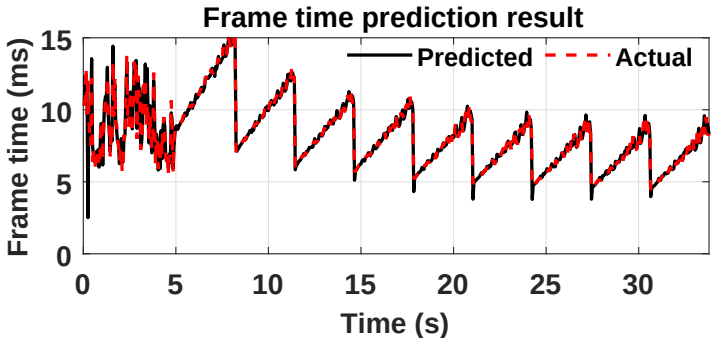

Figure 4.11: Frame time prediction for the RenderingTest app.

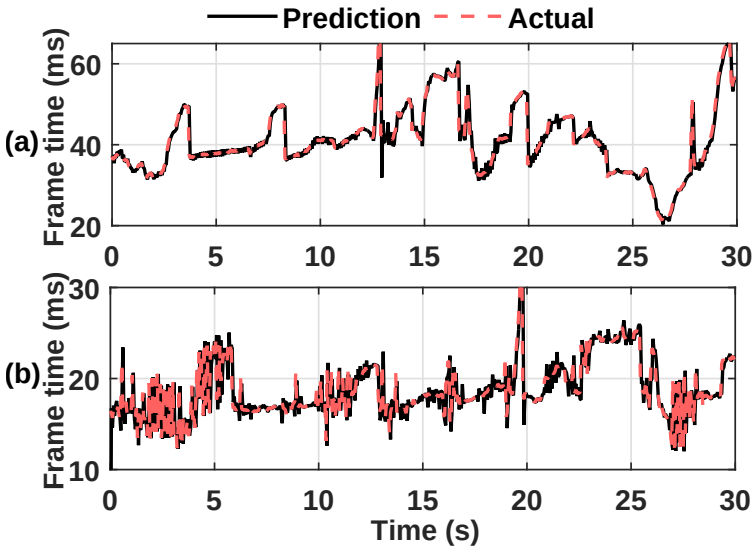

Figure 4.12: Frame time prediction for the 3DMark Ice Storm application running at (a) 200 MHz, (b) 489 MHz.

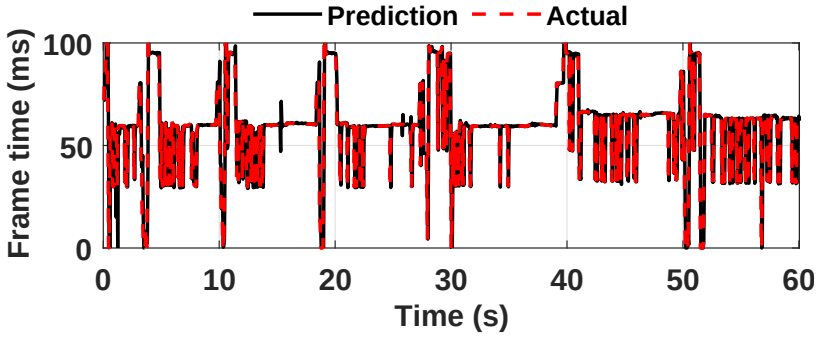

Figure 4.13: Frame time prediction for the BrainItOn application running at 200 MHz.

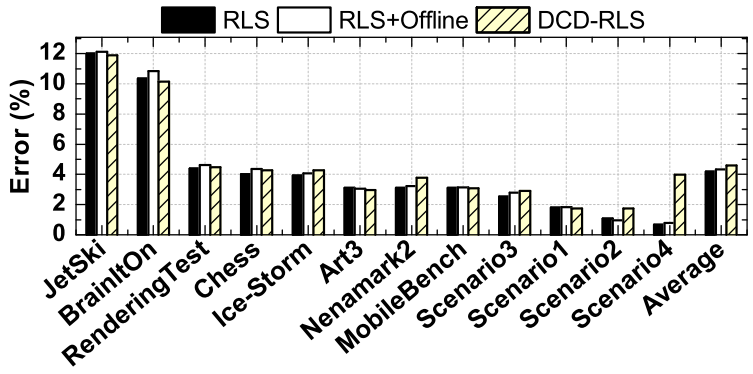

Figure 4.14: Mean absolute percentage errors in the frame time for the Android applications using the three algorithms: RLS, RLS+Offline, and DCD-RLS.

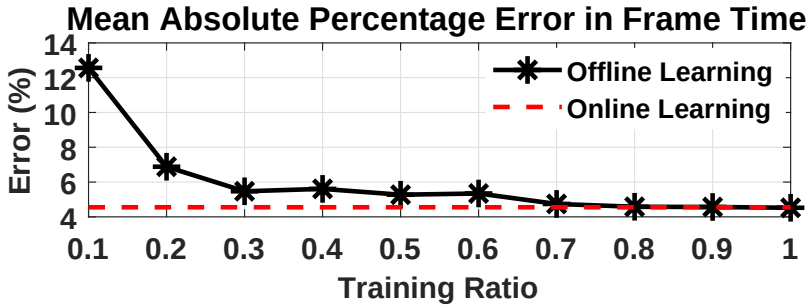

Figure 4.15: Comparison of mean absolute percentage error in frame time for all Android applications combined.

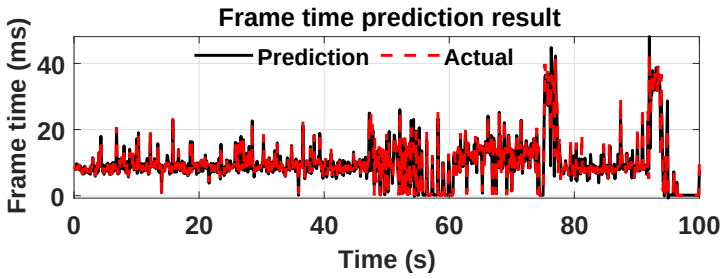

Figure 4.16: Frame time prediction while running YouTube and Chain reaction game running simultaneously on Moto-X smartphone.

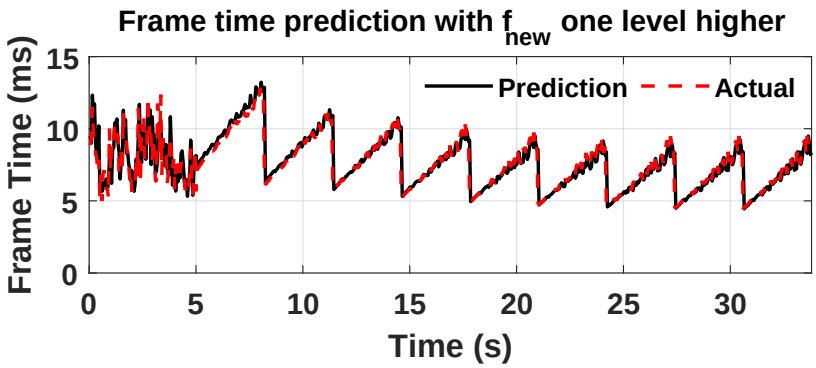

Figure 4.17: Predicted and actual frame times for RenderingTest application when f_{new} is one level higher.

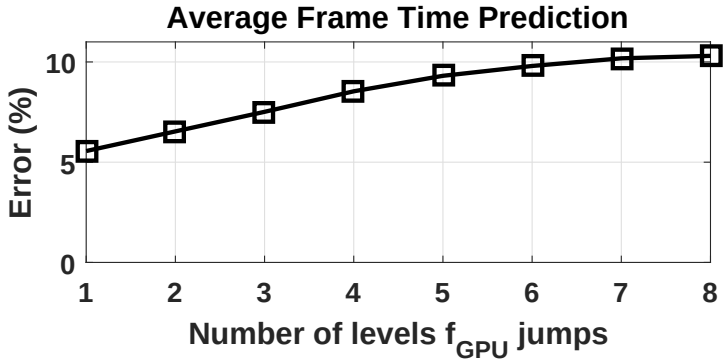

Figure 4.18: Frame time prediction error in RenderingTest application for multiple frequency jumps.

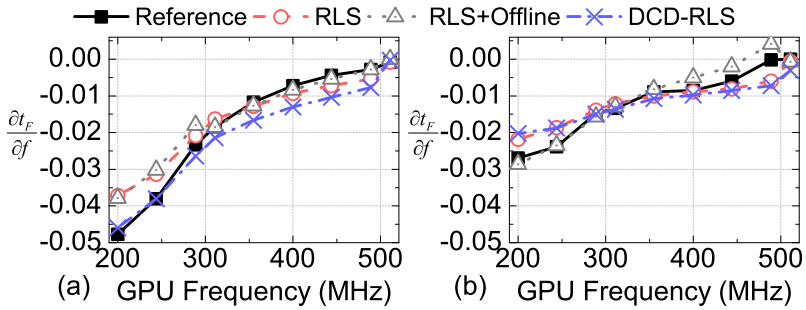

Figure 4.19: Sensitivity of frame time with respect to frequency for (a) RenderingTest and (b) Art3 applications.

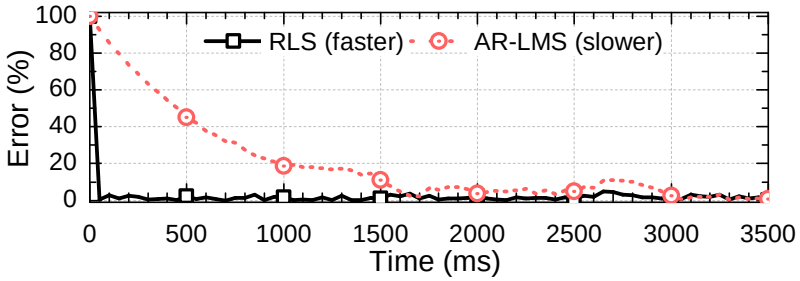

Figure 4.20: The proposed RLS technique converges in only $50ms$ compared to the AR-LMS technique that converges in $1.6s$ for the Icestorm application.

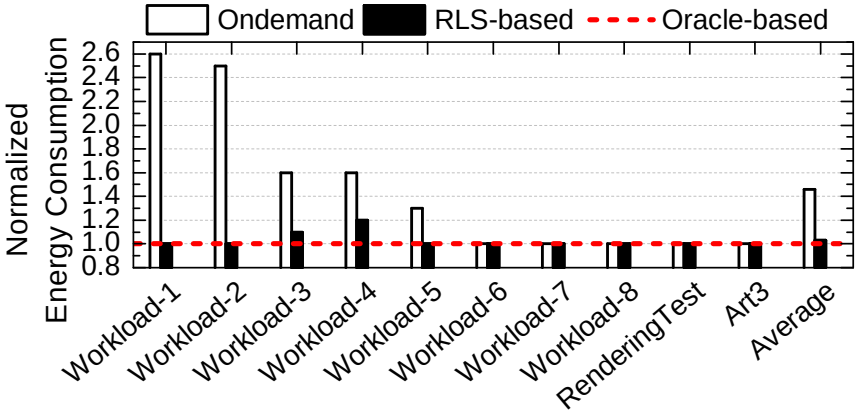

Figure 4.21: Normalized energy consumption of the Ondemand governor and our RLS-based policy normalized to the Oracle-based policy.

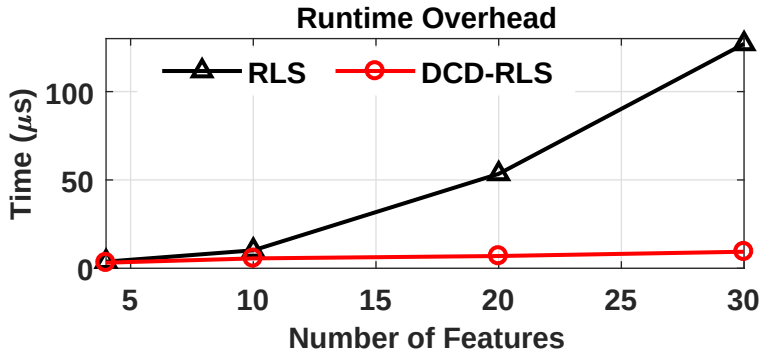

Figure 4.22: Overhead time as a function of the number of features for the RLS and DCD-RLS algorithm.

Chapter 5

STAFF: ONLINE LEARNING WITH STABILIZED ADAPTIVE FORGETTING FACTOR AND FEATURE SELECTION ALGORITHM

5.1 Introduction

Computing systems ranging from mobile platforms to servers run millions of applications, such as games, navigation and browsers. The number and types of these applications are expected to increase further [134]. Sophisticated runtime techniques schedule these applications to the hardware resources and perform dynamic thermal and power management (DTPM) decisions [9, 50, 74, 111, 131]. These techniques need to assess the impact of control variables, such as the operating frequency, to optimize power, performance, energy or other metrics. Therefore, the accuracy of runtime models is critical to meet stringent optimization objectives.

Typically, runtime models are trained offline by employing supervised machine learning, such as batch linear regression [68]. Data used for training involves a set of known application scenarios. There are three main problems with offline approaches. *First*, offline models can only be trained on a limited set of applications available at design time. Hence, these models cannot guarantee a reliable operation for the applications that were not considered during training. *Second*, workloads can be non-stationary, i.e., the statistics like mean and variance of data can change at runtime for different applications [12]. Figure 5.1(a) depicts a non-stationary GPU workload, where both the frame processing time and one of its model coefficient changes as a function of time. Similarly, Figure 5.1(b) shows that the autocorrelation function (ACF) decreases slowly for the entire workload from 0-30 seconds, while ACF

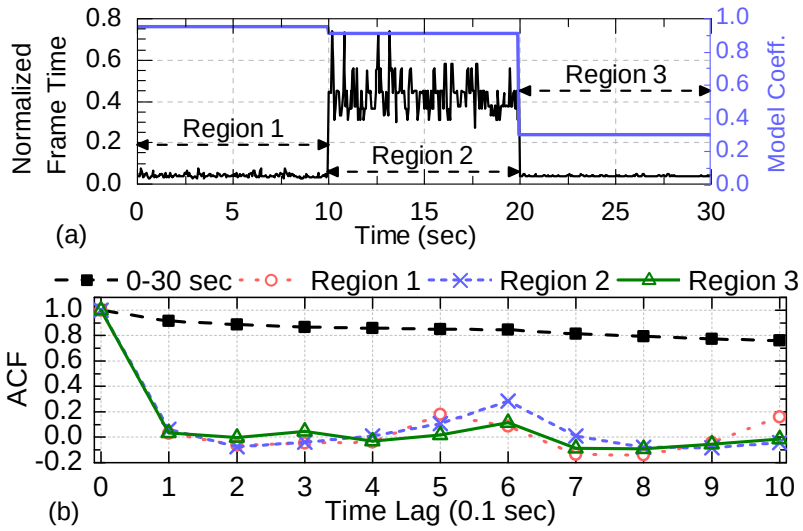

Figure 5.1: A non-stationary GPU workload (a) example and (b) analysis using autocorrelation function (ACF).

for individual phases of the workload ([0,10), [10,20), [20,30] seconds) decays faster. Standard Kwiatkowski-Phillips-Schmidt-Shin (KPSS) and augmented Dickey-Fuller (ADF) tests [49] also confirm that individual regions are stationary, while the entire workload is non-stationary. Consequently, offline models are unsuitable for learning non-stationary workloads as they have fixed coefficients that cannot change at runtime. *Finally*, the optimum set of features that describe the underlying metric change dynamically with the workload. For example, Intel Skylake GPU has 35 hardware counters that can be used as feature inputs to a model [63]. Generally, only a dynamically varying subset of such counters is a good indicator for building a model. Hence, features should be selected at runtime, in contrast to offline models, which rely on a set of features determined at design time.

We present an online learning framework, STAFF, to estimate model coefficients without offline training. STAFF combines *guaranteed stability, online feature selection and adaptive forgetting factor* into a *single computationally efficient runtime*

framework. Online algorithms, such as Recursive Least Squares (RLS), employ exponential forgetting factor to discard old data points and make room for learning from new data points [123]. However, this can result in instability during non-persistently exciting inputs, such as idle period between two workloads [75]. We *guarantee stability* by bounding the correlation matrix of the proposed online algorithm. Moreover, the set of most useful features may change over time, while using all the features leads to over-fitting and increases computational cost. Thus, our framework performs *runtime feature selection* by dynamically computing the correlation of each feature with the output. Finally, the forgetting factor should be adaptive, since old data points need to be forgotten faster during phase transitions, as illustrated in Figure 5.1(a). Our framework also dynamically adapts the forgetting factor to the workload. In summary, STAFF improves the accuracy of runtime models, and reduces their deployment cost on new systems by eliminating offline characterization effort.

In summary, the main contributions of this work are as follows:

- Online feature selection with linear complexity,
- Dynamic forgetting factor methodology that is capable of workload change detection,
- Guaranteed stability while quickly adapting to workload,
- Empirical evaluation on an Intel® Core™ i5 6th generation platform using 17 graphics benchmark scenarios. We achieve up to 6× better accuracy compared to existing techniques.

The remainder of the chapter is organized as follows: Section 5.2 presents an overview of related research. Section 5.3 presents the methodology to develop the proposed STAFF framework. Section 5.4 summarizes the STAFF framework and

shows the complexity analysis. Section 5.5 discusses the experimental results, and Section 5.6 presents the conclusion.

5.2 Related Research

Online learning for power and performance models is relatively new in the power management field. In online learning, model coefficients are estimated at runtime using standard algorithms, such as batch linear regression [111], least mean square (LMS) [28], recursive least squares [40, 50, 83], and an array form of RLS algorithm using QR decomposition [131]. The standard versions of these algorithms are well-known in literature and have severe limitations [123]. In particular, the batch linear regression approach uses costly matrix inversions and large memory in each step, thus providing poor scalability. The standard LMS technique has slower convergence than RLS. Therefore, RLS has turned up as a popular choice for online learning of power and performance models.

RLS is a type of Kalman Filter with fast tracking capabilities for sequential data [123]. Recently, RLS algorithm *without forgetting factor* has been adopted for learning performance models for CPU and GPU [40, 131]. This works well for single workload scenarios, but incorporating variable forgetting factor is crucial for practical applications which switch between different workloads and idle periods [123]. A constant exponential forgetting RLS has been employed for power and performance predictions for CPUs [50, 83]. It is known that all methods that employ exponential forgetting are prone to instability during non-persistently exciting inputs [87]. Kreisselmeier proposed a general class of exponential forgetting RLS algorithm with stability [75]. A special case of Kreisselmeier's algorithm has been shown to guarantee stability under dynamically varying forgetting factor by Milek [87]. However, these algorithms still cannot select features at runtime. A recent approach performs online

feature selection by combining all the features into a single feature [132]. However, this method uses a fixed forgetting factor without stability guarantees. Similarly, Fortescue et al. proposed a variable forgetting factor RLS [33], but their solution cannot neither guarantee stability nor perform online feature selection. In contrast to these approaches [33, 75, 87, 132], STAFF combines adaptive forgetting factor, guaranteed stability and runtime feature selection into a single online learning framework.

An online learning framework also requires a runtime technique to track large changes in workloads, such as transitioning from one application to another. A recent proposal employs Kullback-Leibler (KL) divergence test for this purpose [20]. The authors use only one input variable (the CPU cycles) due to runtime complexity of the KL test. In contrast, we develop a novel low-cost information theoretic approach that utilizes multiple hardware counters.

5.3 STAFF Online Learning Framework

This section introduces first the model templates used for performance modeling. Then, it presents the standard form of RLS with constant forgetting and stabilization. Finally, it presents the online feature selection and adaptive forgetting factor mechanisms.

5.3.1 Model Template

Consider a general linear model with input features $\mathbf{h}_k = [h_{0,k}, h_{1,k}, ..., h_{M-1,k}]^T$ and output y_k at time k,

$$y_k = \mathbf{a}^T \mathbf{h}_k \tag{5.1}$$

where $\mathbf{a} = [a_0, a_1, ..., a_{M-1}]^T$ are model coefficients and $(\cdot)^T$ is the transpose operator. Such linear models are employed by a number of approaches for power and perfor-

mance modeling of the CPU and GPU [40, 83, 136]. For example, a CPU performance model includes predicting cycles per instruction [136]. Similarly, a GPU performance model includes predicting frame processing time [40]. For instance, one can express the change in frame time $\Delta t_{F,k}$ as a function of the GPU frequency f_k and hardware counters $x_{i,k}$ as:

$$\Delta t_{F,k} = a_0 t_{F,k-1}\left(\frac{f_{k-1}}{f_k} - 1\right) + \sum_{i=1}^{M-1} a_i \Delta x_{i,k} \tag{5.2}$$

where $t_{F,k}$ is the frame time at time k, $\mathbf{a}$ are the model coefficients and $[t_{F,k-1}(\frac{f_{k-1}}{f_k} - 1), \Delta x_{1,k}, \Delta x_{2,k}, ..., \Delta x_{M-1,k}]^T = \mathbf{h}_k$ are the features. DTPM and scheduling algorithms can use the model coefficients to make runtime decisions [7, 9]. For example, a_0 denotes the frequency sensitivity of frame time, i.e., it quantifies how frequency affects the frame time. The application phase is memory-bound when $a_0 = 0$. In such as case, lowering the operating frequency will not have any impact on the performance. In contrast, when $a_0 = 1$ the application phase is compute-bound and reducing the frequency will lead to reduction in performance. Note that the GPU performance modeling technique proposed in [40] cannot perform online feature selection and only uses a constant forgetting factor equal to one. Hence, it is not suitable for tracking dynamically varying workloads.

5.3.2 Stability under Exponential Forgetting

The information form of RLS with stabilization and exponential forgetting can estimate the model coefficients $\mathbf{a}$ as follows [75]:

$$\mathbf{R}_k = \lambda \mathbf{R}_{k-1} + \mathbf{h}_k \mathbf{h}_k^T + (1-\lambda)\alpha \mathbf{I}, \qquad \lambda \in (0,1] \text{ and } \alpha \geq 0 \tag{5.3}$$

$$e_k = y_k - \mathbf{a}_{k-1}^T \mathbf{h}_k \tag{5.4}$$

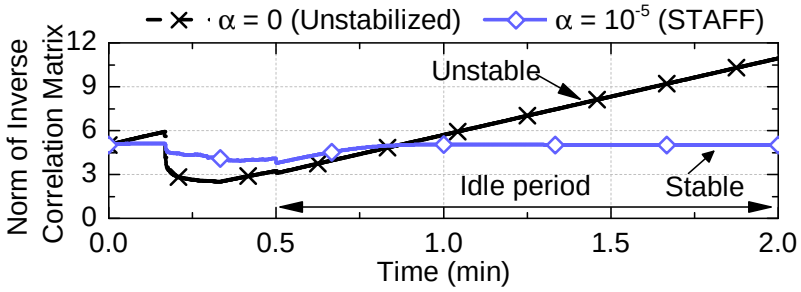

Figure 5.2: The y-axis is in $\log_{10}$scale. ℓ_1 norm of the inverse of the correlation matrix $\mathbf{R}$ shows unstable behavior for $\alpha = 0$ and stability for $\alpha = 10^{-5}$, respectively.

$$\mathbf{a}_k = \mathbf{a}_{k-1} + \mathbf{R}_k^{-1}\mathbf{h}_k e_k \tag{5.5}$$

The correlation matrix $\mathbf{R}_k$ is recursively updated in each iteration. The forgetting factor λ and stabilization factor α are usually equal to one and zero, respectively. The RLS algorithm without forgetting ($\lambda = 1$) has been employed previously for CPU and GPU performance predictions [40, 131]. This leads to infinite memory; hence it accounts for complete history. In turn, this makes RLS slow and unable to adapt to time-varying workloads. Hence, it is crucial to make $\lambda < 1$. The RLS algorithm with constant $\lambda < 1$ has been employed for CPU performance and power predictions [50, 83]. Doing so leads to another problem related to exponential decrease in $\mathbf{R}_k$ or blow-up of $\mathbf{R}_k^{-1}$ under non-persistently exciting inputs, such as no change in input during idle periods in workloads [75, 82]. For example, when $\mathbf{h}_k \to 0$, $\alpha = 0$ and $\lambda < 1$, the recursion in Equation 5.3 leads to a large unbounded increases in $\mathbf{R}_k^{-1}$, as illustrated in Figure 5.2. This can make RLS unstable and sensitive to noise based on the coefficient update in Equation 5.5.

The update of the correlation matrix with $\alpha > 0$ has been theoretically proven to guarantee stability [87]. The stabilizing factor α puts an upper bound on $\mathbf{R}_k^{-1}$. The RLS minimizes a ℓ_2 regularized cost function with regularization parameter μ that is determined using cross-validation [123]. We set $\alpha = \mu$ because the stabilization

factor has a similar effect as the regularization parameter. The correlation matrix and weights are initialized as $\mathbf{R}_0 = \mu\mathbf{I}$ and $\mathbf{a}_0 = \mathbf{0}$, respectively. Figure 5.2 contrasts the blow-up of the inverse of the correlation matrix without stability ($\alpha = 0$) and our result. The STAFF framework with $\alpha = 10^{-5}$ adds an upper bound on the inverse of the correlation matrix, which avoids instability.

5.3.3 Online Feature Selection

State-of-the-art platforms provide a large number of hardware counters that can be used as input features to the power and performance models. For example, the Intel Skylake GPU has 35 hardware counters that can be used as feature inputs to a model [63]. Using all the features leads to large computation overhead due to the inversion of the correlation matrix $\mathbf{R}_k$ in Equation 5.3.

A recently proposed mechanism for runtime feature selection [132] combines all the features into a single combined feature. However, we need to preserve the frequency sensitivity term a_0, since it is important for DTPM decisions. Therefore, instead of using the model $\mathbf{a}^T\mathbf{h}_k$ in Equation 5.1, we estimate an auxiliary model given as follows:

$$y_k = a_0 h_{0,k} + a_c h_{c,k} \tag{5.6}$$

The feature $h_{0,k}$ and coefficient a_0 still represent the frequency term and sensitivity, respectively. The combined feature $h_{c,k}$ is a function of the features $h_{i,k}$ $\forall i \in [1, M-1]$, which captures the workload changes. Hence, our revised model *successfully decouples* the impact of frequency (first term) and workload (second term).

Computing the Combined Feature $h_{c,k}$: A feature that is highly correlated to the output will also be more likely to predict the output compared to the features that are less correlated. Thus, we start with a number of single-input affine functions

between the output y_k and each of the features $h_{i,k}$,

$$y_k = b_i + c_i h_{i,k} \qquad \forall i \in [1, M-1] \tag{5.7}$$

where the model coefficients b_i and c_i are estimated by applying two-input RLS described by Equations 5.3-5.5. Then, we compute the Pearson correlation coefficients, $\rho_{i,k}$ between the output y_k and each feature $h_{i,k}$ as follows:

$$\rho_{i,k} = \frac{\sigma_{h_{i,k}}}{\sigma_{i,y_k}} c_{i,k}, \text{where } -1 \leq \rho_{i,k} \leq 1 \quad \forall i \in [1, M-1] \tag{5.8}$$

The standard deviations $\sigma_{h_{i,k}}$and σ_{i,y_k} are computed recursively by employing forgetting factors [32]. Note that each model can employ a different forgetting factor λ; therefore, the standard deviations σ_{i,y_k} may not be same for each model output. Then, we combine the correlation coefficients to find the likelihood $\pi_{i,k}$ of a feature $h_{i,k}$ to predict the output y_k as follows:

$$\pi_{i,k} = \frac{\rho_{i,k}^2}{\sum_{i=1}^{M-1}(\rho_{i,k}^2)}, \text{where } 0 \leq \pi_{i,k} \leq 1 \qquad \forall i \in [1, M-1] \tag{5.9}$$

Note that the likelihoods $\pi_{i,k}$ sum up to one. Finally, the combined feature is expressed as the weighted sum of original features $h_{i,k}$:

$$h_{c,k} = \sum_{i=1}^{M-1} \pi_{i,k} h_{i,k} \qquad \forall i \in [1, M-1] \tag{5.10}$$

Now that $h_{0,k}$ and $h_{c,k}$ are known, we estimate the model coefficients a_0 and a_c in Equation 5.6 by applying the two-input RLS described by Equations 5.3-5.5.

Figure 5.3 shows the comparison between the offline feature selection algorithm and the STAFF framework. STAFF tracks the reference value a_0 without the erroneous troughs by changing the features dynamically at runtime. The offline selection algorithm gives poorer results due to overfitting. There are two additional advantages

of STAFF: First, the implementation is very low cost (see Section 5.4). Second, it provides the flexibility to select desired features, such as $h_{0,k}$, while merging other features, such as $h_{i,k}$ $\forall i \in [1, M-1]$, to reduce the runtime complexity.

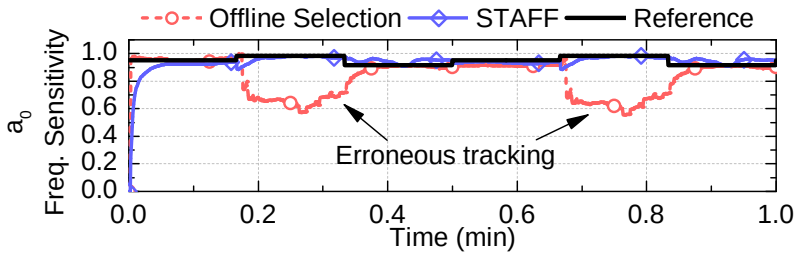

Figure 5.3: Online STAFF framework has superior tracking performance to offline feature selection.

5.3.4 Adaptive Forgetting Factor

A constant forgetting factor is not desirable, because the amount of history that needs to be forgotten changes dynamically. Therefore, we employ an auto-tuning technique to adapt λ to a wide variety of workloads [33]. This technique preserves the information content in each sample of RLS by adapting the forgetting factor as a function of the inputs $\mathbf{h}_k$, inverse of correlation matrix $\mathbf{R}_{k-1}^{-1}$, RLS error e_{k-1} and an initial estimate of the information content $\Sigma_0 = \sigma^2 N_0$. In these expressions, σ^2 is the process noise estimate and $N_0 = \frac{1}{1-\lambda_0}$ is the initial asymptotic memory length.

$$\lambda_k = 1 - \frac{(1 - \mathbf{h}_k \mathbf{R}_{k-1}^{-1} \mathbf{h}_k^T) e_{k-1}^2}{\Sigma_0} \tag{5.11}$$

λ_k is bounded using a lower bound λ_{LB} and an upper bound λ_{UB} such that $0 << \lambda_{LB} < \lambda_k < \lambda_{UB} \leq 1$. However, Equation 5.11 *cannot track fast changes*, such as switches from one application to another. In these cases, information stored in the correlation matrix need to be forgotten faster. Therefore, *we propose a novel*

technique for workload change detection based on Shannon's entropy.

Workload Change Detection: A large change in workload, such as a new type of frame, can vary the model output y_k and parameters of Equation 5.6. Information about the type of frame is embedded in the features $h_{i,k}\ \forall i \in [1, M-1]$ of the hardware counters, as mentioned in Section 5.3.3. We use the likelihoods $\pi_{i,k}$, of obtaining y_k from $h_{i,k}$, to compute the entropy H_k as follows:

$$H_k = -\sum_{i=1}^{M-1} \pi_{i,k} log_b(\pi_{i,k}) \tag{5.12}$$

where the logarithm base $b = M - 1$, such that the entropy is normalized. The logarithm function can be implemented using a fast binary algorithm [140]. When the likelihoods $\pi_{i,k}$ are all equal, the hardware counters have the most disorder (highest entropy), i.e., no one feature is more important than the other. Conversely, when the likelihoods are not equal, then the entropy is smaller. Figure 5.4 illustrates the change in entropy for two features. In region 1, the likelihoods of feature 1 and feature 2 are $\pi_1 = 0.8$ and $\pi_2 = 0.2$, respectively. Then, the workload changes from region 1 to region 2 where $\pi_1 = 0.1$ and $\pi_2 = 0.9$. Due to the recursive computations involved in finding $\pi_{i,k}$, the likelihoods do not instantly jump from small to large values or vice versa. During the transition, the likelihoods slowly change and become equal ($\pi_1 = 0.5$ and $\pi_2 = 0.5$), leading to highest entropy. Consequently, the information content H_k becomes close to one whenever the workload changes significantly. In this condition, we make the lower bound $\lambda_{LB} = 0.001$, which is still guaranteed to be stable. If the adaptive forgetting mechanism chooses the lower bound λ_{LB}, a resetting effect on the correlation matrix is achieved, leading to very fast tracking of workloads. Based on our experiments, we find this method is suitable for capturing all workload changes. Figure 5.5 shows that variable forgetting performed by STAFF is more desirable than constant forgetting ($\lambda = 0.99$ and $\lambda = 0.9$) algorithms that

lead to poor tracking of the frequency sensitivity a_0.

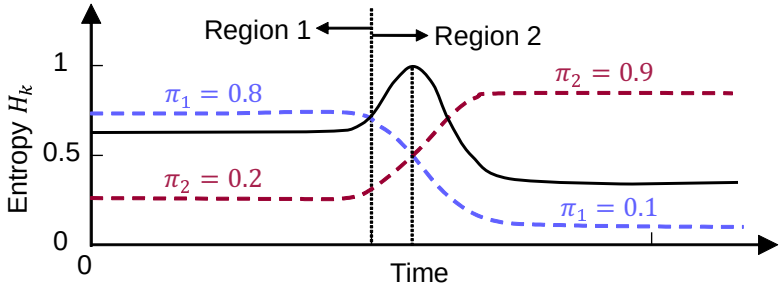

Figure 5.4: Illustration of the entropy-based change detection. The solid-line shows the entropy, while the and dashed-lines show the likelihoods of feature 1 and feature 2, respectively.

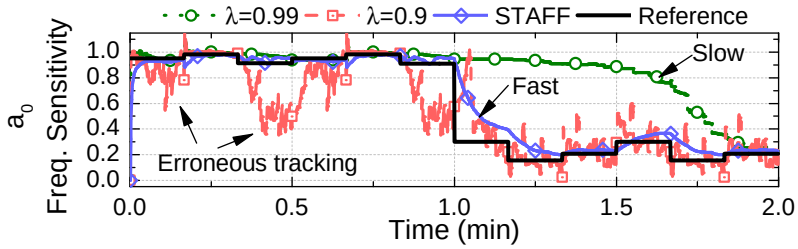

Figure 5.5: The STAFF framework adapts much faster to the new workload compared to $\lambda = 0.99$. In addition, it does not possess the local erroneous tracking of a_0 caused by $\lambda = 0.9$.

5.4 Summary and Complexity Analysis

Figure 5.6 summarizes one iteration of the STAFF framework. We first achieve stability by employing the term $\alpha > 0$ in Equation 5.3. Next, we perform online feature selection by combining features at runtime based on their correlation coefficient to the output using Equations 5.9 and 5.10. This requires running a total of $M - 1$ two-input RLS for estimating the composite feature in Equation 5.6. Then,

we dynamically compute the forgetting factor using Equations 5.11 and 5.12. Subsequently, we learn the model coefficients in 5.7 using one more two-input RLS. Finally, we compute the frequency sensitivity by using a moving average with forgetting on the estimated value of a_0 in Equation 5.6. This is done to remove edge effect overshoots caused by sharp tracking during workload changes.

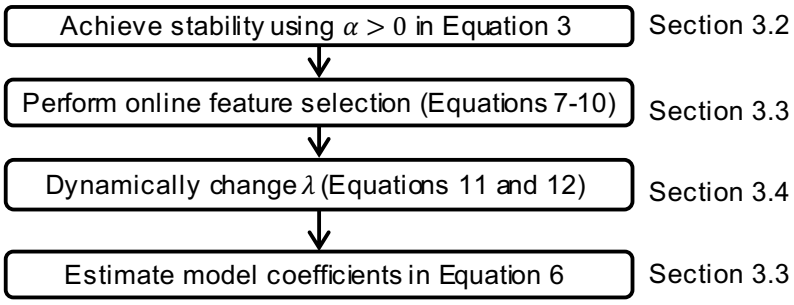

Figure 5.6: Summary of the STAFF algorithm.

Complexity Analysis: Model estimation algorithms must be very low-cost as they are used with DTPM algorithms. Table 5.1 presents the number of scalar operations for each iteration of the proposed STAFF algorithm and a competitive baseline algorithm that employs stabilized exponential variable forgetting (SEF) with offline feature selection [87]. The SEF algorithm requires inverting the $\mathbf{R}$ matrix in Equation 5.5, whose computational complexity is $O(M^3)$. In contrast, the STAFF approach requires only M simple 2×2 matrix inversions of $\mathbf{R}$, which only linearly increases complexity. Therefore, the STAFF algorithm provides better scalability than the SEF algorithm. Of note, the unstabilized version of the standard RLS algorithm has a computational complexity of $O(M^2)$, since it is possible to apply the matrix inversion lemma [123].

In practice, the SEF algorithm is used in conjunction with offline feature selection (Offline_FS) methods that reduce the number of input features. The complexity of the SEF algorithm changes with the number of features M. In contrast, the STAFF

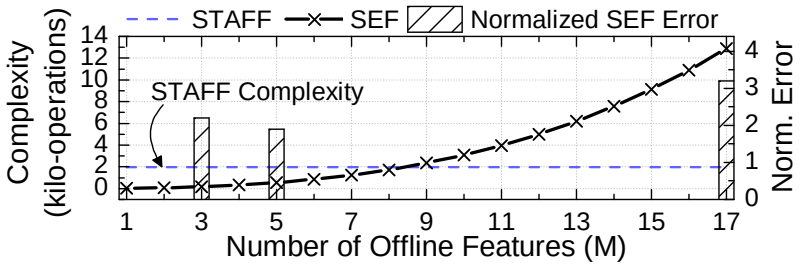

Figure 5.7: STAFF framework has 3.2× lower error (right axis) in frequency sensitivity, and 6.5× lower complexity compared to the SEF algorithm for $M = 17$. The errors are computed using the non-stationary workload employed in Section 5.5.2

.

framework uses all the features ($M = 17$) while keeping a constant complexity, as shown in Figure 5.7. We observe that the complexity of SEF is 6.5× larger than STAFF for $M = 17$. At this complexity, SEF leads to 3.2× larger mean absolute error in predicting frequency sensitivity compared to STAFF. Even though SEF has lower complexity with $M = 3$ and $M = 5$, this leads to 2.2× and 1.9× higher error, respectively. Thus, STAFF achieves higher accuracy by using online feature selection with only linear complexity.

Table 5.1: The number of algebraic operations in the SEF and STAFF algorithms in each iteration.

Alg.	**Mul**	**Add/Sub**	**Div**	**Big O**
SEF	$M^3 + 4M^2 + 7M + 5$	$M^3 + 4M^2 + 2M + 3$	$2M^2$ - M	M^3
STAFF	60M - 6	55M - 12	2M + 2	M

5.5 Experiments

5.5.1 Experimental Setup

In our experiments we utilize the Intel® Core™ i5 6th generation platform with Microsoft Windows 10 OS. We communicate with firmware using the kernel drivers to read the performance counters, frame time and frequency of the GPU with the help of our custom user-space routines in C. By employing AutoHotKey, we create 17 real application scenarios (recorded user interactions) and synthetic application scenarios (shuffled pre-recorded frames) using representative set of graphic applications, such as, 3Dmark, Angry Birds, Angry Bots, Jungle Run, Shark Dash, and Fruit Ninja, to evaluate our approach. We estimate the noise variance of frame time in our data set as 0.1 ms^2 to use in Equation 5.11. *STAFF source code is available at* ***http://elab.engineering.asu.edu/public-release/***.

Baseline Algorithms: We compare our approach to several baseline algorithms, shown in Table 5.2. One approach employs RLS with Offline_FS and constant $\lambda = 1$ [40], and another with constant $\lambda < 1$ [50, 83]. *The Offline_FS and SEF [75, 87] algorithms perform costly offline feature selection instead of the low cost online feature selection performed by the proposed STAFF approach.* The Offline_FS algorithm selects the features offline using Lasso regression over 17 application scenarios with 10-fold cross-validation. In contrast, the Offline_FS* learns the features using only the test workloads, thus it is the best baseline when the test data is known in advance. We also compare against SEF with All-features that uses all the hardware counters read from the platform.

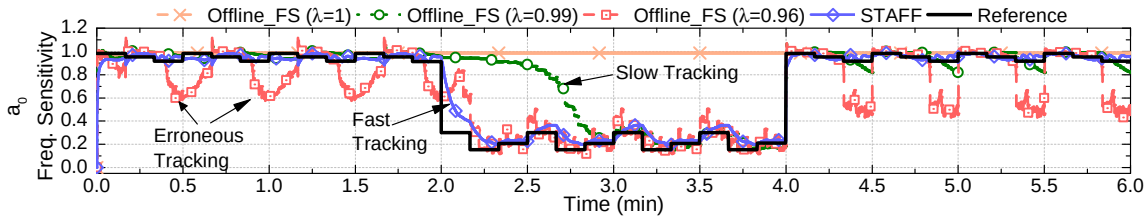

Figure 5.8: Comparison of the STAFF framework against *constant* forgetting factor approaches.

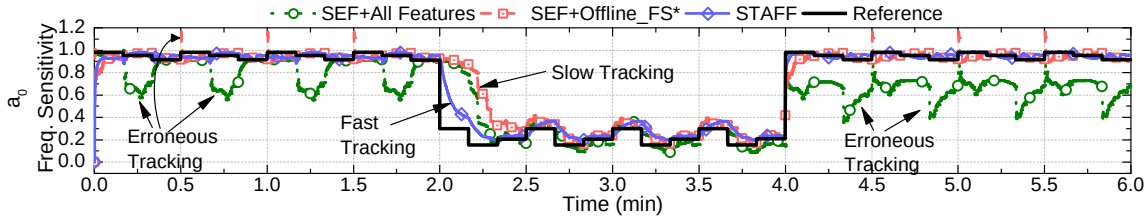

Figure 5.9: Comparison of the STAFF framework against *adaptive* forgetting factor approaches.

5.5.2 *Evaluating the GPU Performance Model*

We combine a subset of 17 single applications to compose a real world scenario that transitions between different applications. Under the composed workload, the baseline algorithms perform very poorly, even though they can track individual applications successfully. Figure 5.8 shows such a workload that includes three frequency sensitivity patterns. First, three applications with high sensitivity ($a_0 \approx 1$) of duration 10 seconds each are run during the region 0-2 minutes. Then, three other applications with low sensitivity ($a_0 \approx 0$) are run during region 2-4 minutes. Finally, three applications with high sensitivity are run from 4-6 minutes. *Clearly, this workload is non-stationary because the reference frequency sensitivity (model coefficient a_0) is time varying.* This workload captures all major transitions in the sensitivity, such as, high to low.

Benefits of Adaptive Forgetting: Figure 5.8 shows the comparison between three

Table 5.2: Summary of the baseline algorithms and the proposed STAFF framework.

Algorithm	# Features	Variable λ & $\alpha > 0$	Complexity
Offline_FS ($\lambda = 1$)	3	–	$O(M^2)$
Offline_FS ($\lambda < 1$)	3	–	$O(M^2)$
SEF+All features	17	✓	$O(M^3)$
SEF+Offline_FS*	5	✓	$O(M^3)$
STAFF	17	✓	$O(M)$

RLS algorithm baselines and the proposed STAFF framework. A basic RLS approach with Offline_FS ($\lambda = 1$) cannot track the workload between 2 minutes to 4 minutes. This is because past information learned from 0 to 2 minutes is not forgotten. The Offline_FS ($\lambda = 0.99$) is very slow to track and takes about 1 minute to go from high to low frequency sensitivity. A lower forgetting factor ($\lambda = 0.96$), may seem a natural choice to track better. While this algorithm tracks faster in the transition at 2 minutes, there are now local troughs at several location, such as 0.5, 1, and 4.5 minutes. Therefore, an adaptive forgetting scheme is better as it forgets less in the high sensitivity regions and forgets more during the transition regions. The STAFF approach performs accurate tracking of the sensitivity in all the time intervals. The transition interval from high to low sensitivity is about 3 seconds, which is 14× faster than $\lambda = 0.99$. This happens, since the forgetting factor becomes low ($\lambda = 0.001$) when a workload change is detected. Otherwise, λ varies between the bounds $\lambda_{LB} = 0.96$ to $\lambda_{UB} = 0.99$.

Benefits of Online Feature Selection: Figure 5.9 shows the comparison of the proposed STAFF framework with SEF baseline algorithms that can dynamically change the forgetting factor with stabilization, but without online feature selection. SEF algorithm that uses all features fail to provide good accuracy over the entire

workload. In particular, we find that using all the features leads to over-fitting and poor estimates of the frequency sensitivity during 0-2 minute and 4-6 minute regions. Similarly, we find that SEF algorithm that uses the best set of features selected offline (Offline_FS*) performs better than using all the features. However, it has local erroneous peaks and it is slow to converge during the application scenario changes in comparison to the STAFF framework. Table 5.3 shows the index of the three most frequently used features that are critical during different workload regions. Clearly, the most important features change with time. Fixed feature algorithms, such as SEF, cannot track the frequency sensitivity with high accuracy.

Error Analysis: Figure 5.10 shows the relative region-wise errors in frequency sensitivity estimate of the algorithms with respect to the best baseline SEF+Offline_FS* algorithm. In this figure, we exclude the baseline Offline_FS ($\lambda = 0.99$) since it is clearly worse than Offline_FS ($\lambda = 0.96$) during workload transitions. The lowest error for STAFF is obtained in the workload transition period of 2-3 minutes due to our entropy based workload change detection mechanism. The Offline_FS ($\lambda = 1$) algorithm can show up to 11.9× higher error during the low frequency sensitivity regions of 2-4 minutes. Similarly, the Offline_FS ($\lambda = 0.96$) and SEF algorithms with all features show up to 6.5× and 8.5× higher error during the high frequency sensitivity region of 5-6 minutes. The error in STAFF is marginally higher than these two algorithms in the period 3-4 minutes. However, it is still within one standard deviation of the frequency sensitivity reference value. The STAFF algorithm outperforms all the baselines during all other intervals. In summary, for the entire workload (0-6 minutes), the STAFF framework provides 6×, 2.2×, 3.2×, and 1.9× better accuracy than the Offline_FS ($\lambda = 1$), Offline_FS ($\lambda = 0.96$), SEF with all features, and SEF with Offline_FS* baseline algorithms, respectively.

Workload Change Detection Analysis: Figure 5.11 shows the correlation co-

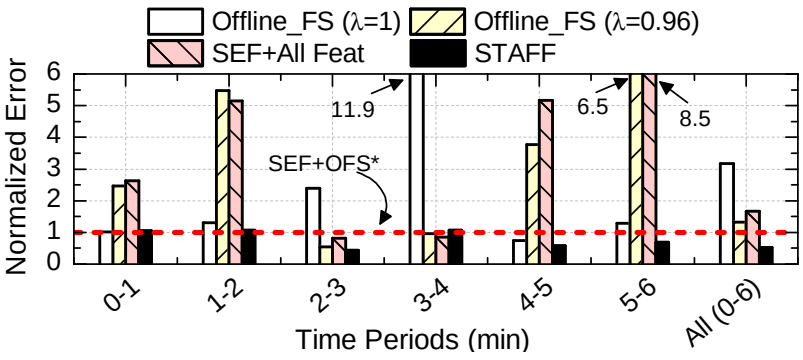

Figure 5.10: Normalized RMS error in the frequency sensitivity estimates of the algorithms in different workload regions. Errors are normalized with respect to the best baseline approach (SEF+Offline_FS*).

efficients, likelihoods, and entropy of the hardware counter features for the STAFF algorithm. We show a subset of samples around 2 minutes time of the test workload (shown in Figure 5.9) for clarity. When the workload changes at time 2 minutes, the absolute value of the correlation coefficients for the 16 hardware counter features becomes high and closer to each other in the subsequent time interval, as shown in Figure 5.11(a). This leads to the likelihood of all the hardware counter features to become equal ($1/16 = 0.0625$) in Figure 5.11(b). Consequently, the entropy becomes close to one in Figure 5.11(c), indicating that the workload has changed.

Table 5.3: Three most correlated features used by the STAFF in different time regions of the workload in Figure 5.9.

Time in Fig. 5.9	0-2 min	2-4 min	4-6 min	All (0-6 min)
Feature #1	h_1	h_{14}	h_1	h_1
Feature #2	h_2	h_7	h_2	h_2
Feature #3	h_4	h_6	h_4	h_6

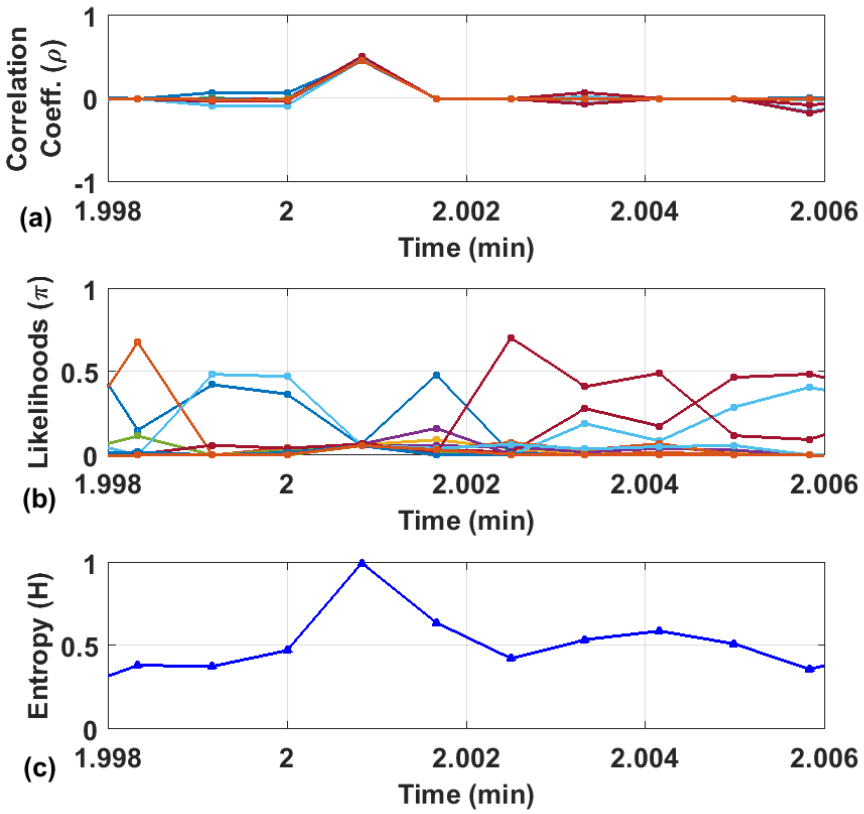

Figure 5.11: Analysis of correlation coefficients, likelihoods, and entropy of the hardware counter features for STAFF algorithm at 2 minutes time of Figure 5.9. (a) The correlation coefficients become equal in the time interval immediately after a workload change occurs at time 2 minutes. (b) The likelihood values become equal in the same time interval. (c) The entropy for the set of the hardware counter features also changes and peaks in the same time interval.

5.5.3 *Faster Convergence for STAFF*

In this section, we discuss a faster version of the STAFF algorithm, called Fast-STAFF, that is able to adapt to the reference frequency sensitivity in shorter duration than STAFF algorithm. In STAFF algorithm we applied moving average filter to the coefficient a_0 to obtain the frequency sensitivity, as discussed in Section 5.4. We chose a variable filter width based on the forgetting factor employed for the main RLS of Equation 5.6. This helps in reducing an extra parameter in the algorithm. However, it causes the STAFF algorithm to be slower at the transitions of high frequency sensitivity changes, such as at time 2 minutes in Figure 5.9. In particular, at this largest workload transition we observe a convergence time of 3.75 seconds in the STAFF algorithm. Instead of over-provisioning the STAFF algorithm with a variable width moving average filter, we fix the filter width to reduce the convergence penalty. Our experiments reveal a good choice for the filter-width is 5, which indicates a 4 Hz filter frequency for our data sampled at 20 Hz frequency. The Fast-STAFF algorithm provides 1.85 second convergence time at the largest workload transition, as shown in Figure 5.12, which is about 2× faster.

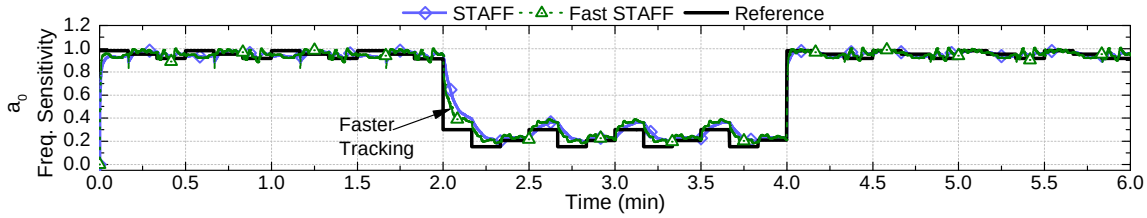

Figure 5.12: Comparison of the STAFF and Fast-STAFF algorithm.

5.6 Conclusion

In this chapter, we present a novel framework, called STAFF, that guarantees stability, performs online feature selection with linear complexity, and dynamically

changes the forgetting factor. We evaluate it by predicting the frequency sensitivity of a graphics unit in a commercial Intel platform. The framework provides fast tracking with up to 6× improvement in the prediction accuracy compared to existing state-of-the-art techniques.

Chapter 6

CONCLUSION

Power management has become crucial for heterogeneous systems that integrates many processing elements, such as CPU cores, GPU, video, image, and audio processors. Dynamic thermal and power management algorithms address this problem by putting a subset of the processing elements or shared resources to sleep states, or throttling their frequencies. However, an adhoc approach could easily cripple the performance, if it slows down the performance-critical processing element. Furthermore, mobile platforms run a wide range of applications with time varying workload characteristics, unlike early generations, which supported only limited functionality. As a result, there is a need for adaptive power and performance management approaches that consider the platform as a whole, rather than focusing on a subset. Towards this need, our first contribution is a dynamic power management technique for a recently introduced single ISA big.LITTLE heterogeneous CPU system [51]. Our experiments show an average increase of 93%, 81% and 6% in performance per watt compared to the interactive, ondemand and powersave governors, respectively. While the CPU is one of the most important components of a SoC, a number of mobile applications, such as games critically depend on the GPU for rendering. Therefore, in our second contribution, we focus on power management of the CPU and GPU together for graphics workloads. The experiments on an Intel Baytrail platform [62] show up to 15% increase in average frame rate compared to the default power allocation algorithms. Our third contribution targets integrated GPUs, since they have become an indispensable component of mobile processors due to the increasing popularity of graphics applications. The GPU frequency is a key factor both in application throughput and

mobile processor power consumption under graphics workloads. Therefore, dynamic power management algorithms have to assess the performance sensitivity to the GPU frequency accurately. Since the impact of the GPU frequency on performance varies rapidly over time, there is a need for online performance models that can adapt to varying workloads. To address this, we propose a frame time model that does not rely on any parameter learned offline. Our experiments on the Intel Minnowboard MAX platform running common GPU benchmarks show that the mean absolute percentage error in frame time and frame time sensitivity prediction are 4.2% and 6.7%, respectively. Finally, online learning of power and performance models require efficient online learning algorithms that can adapt to multiple applications, determine the important features at runtime and lead to stable solutions. To address this need, we develop a novel online learning algorithm, STAFF that performs online feature selection and adapts to non-stationary workloads using dynamically varying forgetting factor with stability.

This dissertation summarizes our contributions that aid the power management of heterogeneous mobile platforms. More precisely, our specific contributions are as follows:

- A framework to dynamically select the Pareto-optimal frequency and active cores for the heterogeneous CPUs, such as ARM big.LITTLE architecture [46],
- A dynamic power budgeting approach for allocating optimal power consumption to the CPU and GPU using performance sensitivity models for each PE [47],
- An adaptive GPU frame time sensitivity prediction model to aid power management algorithms [40, 109].
- An online learning algorithm with stabilized adaptive forgetting factor and runtime feature selection capabilities [39].

REFERENCES

[1] Aalsaud, A. *et al.*, "Power–Aware Performance Adaptation of Concurrent Applications in Heterogeneous Many-Core Systems", in "Proc. of the Intl. Symp. on Low Power Elec. and Design", pp. 368–373 (2016).

[2] Akenine-Möller, T., E. Haines and N. Hoffman, *Real-time rendering* (CRC Press, 2008).

[3] Amdahl, G. M., "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities", in "Proc. of ACM Spring Joint Computer Conf.", (1967).

[4] AnandTech, "ARM's Mali Midgard Architecture Explored", `https://www.anandtech.com/show/8234/arms-mali-midgard-architecture-explored/4` (2017).

[5] App Tornado, "App Brain", `http://www.appbrain.com/` (2016).

[6] Apple Inc., https://support.apple.com/en-us/HT201678 (2017).

[7] Ayoub, R. Z. *et al.*, "OS-level Power Minimization under Tight Performance Constraints in General Purpose Systems", in "Proc. of the Intl. Symp. on Low-power Elec. and Design", pp. 321–326 (2011).

[8] Benini, L., A. Bogliolo and G. De Micheli, "A Survey of Design Techniques For System-Level Dynamic Power Management", IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **8**, 3, 299–316 (2000).

[9] Bhat, G., G. Singla, A. K. Unver and U. Y. Ogras, "Algorithmic optimization of thermal and power management for heterogeneous mobile platforms", IEEE Trans. on Very Large Scale Integration Syst. **26**, 3, 544–557 (2018).

[10] Bhat, G. *et al.*, "Multi-Objective Design Optimization for Flexible Hybrid Electronics", in "Proc. of Int. Conf. on Comput. Aided Design", (2016).

[11] Bienia, C., S. Kumar, J. P. Singh and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications", in "Proc. of the Intl. Conf. on Parallel Arch. and Compilation Tech.", pp. 72–81 (2008).

[12] Bogdan, P. and R. Marculescu, "Non-stationary traffic analysis and its implications on multicore platform design", IEEE Trans. on Computer-Aided Design of Integrated Circuits and Syst. **30**, 4, 508–519 (2011).

[13] Bogdan, P., R. Marculescu, S. Jain and R. T. Gavila, "An Optimal Control Approach to Power Management for Multi-Voltage and Frequency Islands Multiprocessor Platforms under Highly Variable Workloads", in "Proc. of the Intl. Symp. on Networks on Chip", pp. 35–42 (2012).

[14] Carroll, A. and G. Heiser, "An Analysis of Power Consumption in a Smartphone", in "USENIXATC", (2010).

[15] Chen, W.-M., S.-W. Cheng, P.-C. Hsiu and T.-W. Kuo, "A User-Centric CPU-GPU Governing Framework for 3D Games on Mobile Devices", in "Proc. of the Intl. Conf. on Computer-Aided Design", pp. 224–231 (2015).

[16] Chen, X. *et al.*, "Dynamic Voltage and Frequency Scaling for Shared Resources in Multicore Processor Designs", in "Proc. of the Design Autom. Conf.", p. 114 (2013).

[17] Cochran, R., C. Hankendi, A. K. Coskun and S. Reda, "Pack & Cap: Adaptive DVFS and Thread Packing Under Power Caps", in "Proc. of the Intl. Symp. on Microarch.", pp. 175–185 (2011).

[18] Cortex, A., "A15 MPCore Processor Technical Reference Manual", ARM Holdings PLC **24** (2013).

[19] Coskun, A. K., T. S. Rosing and K. Whisnant, "Temperature Aware Task Scheduling in MPSoCs", in "Proc. of the Conf. on Design, Autom. and Test in Europe", pp. 1659–1664 (2007).

[20] Das, A., G. V. Merrett, M. Tribastone and B. M. Al-Hashimi, "Workload Change Point Detection for Runtime Thermal Management of Embedded Systems", Trans. on Comp.-Aided Des. of Int. Circuits and Sys. **35**, 8, 1358–1371 (2016).

[21] David, R., P. Bogdan, R. Marculescu and U. Ogras, "Dynamic Power Management of Voltage-frequency Island Partitioned Networks-on-Chip using Intel's Single-chip Cloud Computer", in "Proc. of the Intl. Symp. on Networks on Chip", pp. 257–258 (2011).

[22] de Melo, A. C., "The New Linux Perf Tools", in "Linux Kongress", vol. 18 (2010).

[23] Del Sozzo, E. *et al.*, "Workload-aware Power Optimization Strategy for Asymmetric Multiprocessors", in "Proc. of the Design, Auto. & Test in Europe Conf. & Exhib.", pp. 531–534 (2016).

[24] Dev, K., A. N. Nowroz and S. Reda, "Power Mapping and Modeling of Multi-Core Processors", in "Proc. of the Intl. Symp. on Low Power Electronics and Design", pp. 39–44 (2013).

[25] Dev, K. and S. Reda, "Scheduling Challenges and Opportunities in Integrated CPU+GPU Processors", in "Proc. of the Symp. of Embedded Systems For Real-time Multimedia", pp. 1–6 (2016).

[26] Dhiman, G. and T. S. Rosing, "System-Level Power Management Using Online Learning", IEEE Trans. Comput.-Aided Design Integr. Circuits and Syst. **28**, 5, 676–689 (2009).

[27] Dietrich, B. and S. Chakraborty, "Lightweight Graphics Instrumentation for Game State-Specific Power Management in Android", Multimedia Systems **20**, 5, 563–578 (2014).

[28] Dietrich, B. *et al.*, "LMS-based Low-complexity Game Workload Prediction for DVFS", in "Proc. of the Intl. Conf. on Comp. Design", pp. 417–424 (2010).

[29] Donyanavard, B., T. Mück, S. Sarma and N. Dutt, "SPARTA: Runtime Task Allocation for Energy Efficient Heterogeneous Many-cores", in "Proc. of the Intl. Conf. on Hardware/Software Codesign and Sys. Syn.", p. 27 (2016).

[30] Faith, R., "The Direct Rendering Manager: Kernel Support for the Direct Rendering Infrastructure", (1999).

[31] Farrar, D. E. and R. R. Glauber, "Multicollinearity in Regression Analysis: the Problem Revisited", The Review of Economic and Statistics, JSTOR pp. 92–107 (1967).

[32] Finch, T., "Incremental Calculation of Weighted Mean and Variance", University of Cambridge **4**, 11–5 (2009).

[33] Fortescue, T., L. S. Kershenbaum and B. E. Ydstie, "Implementation of Self-Tuning Regulators with Variable Forgetting Factors", Automatica **17**, 6, 831–835 (1981).

[34] Friedman, J., T. Hastie and R. Tibshirani, *The Elements of Statistical Learning*, vol. 1 (Springer Series in Statistics, Berlin, 2001).

[35] Gandhi, A., M. Harchol-Balter, R. Das, J. O. Kephart and C. Lefurgy, "Power Capping Via Forced Idleness", in "Proc. of Workshop on Energy-Efficient Design", (2009).

[36] Ghasemazar, M., E. Pakbaznia and M. Pedram, "Minimizing Energy Consumption of a Chip Multiprocessor Through Simultaneous Core Consolidation and DVFS", in "Proc. of the Intl. Symp. on Circuits and Systems", pp. 49–52 (2010).

[37] Google, O., "Android Jelly Bean", `http://www.android.com/versions/jelly-bean-4-2/` (2017).

[38] Gu, Y., S. Chakraborty and W. T. Ooi, "Games are up for DVFS", in "Proc. of the Design Automation Conf.", pp. 598–603 (2006).

[39] Gupta, U., M. Babu, R. Ayoub, M. Kishinevsky, F. Paterna and U. Y. Ogras, "STAFF: Online Learning with Stabilized Adaptive Forgetting Factor and Feature Selection Algorithm", in "Proc. of Design Autom. Conf.", p. 6 (2018 (To appear)).

[40] Gupta, U., J. Campbell, U. Y. Ogras, R. Ayoub, M. Kishinevsky, F. Paterna and S. Gumussoy, "Adaptive Performance Prediction for Integrated GPUs", in "Proc. of the Intl. Conf. on Computer-Aided Design", p. 61 (2016).

[41] Gupta, U., S. Jain and U. Y. Ogras, "Can Systems Extend to Polymer? SoP Architecture Design and Challenges", in "Proc. of Int. SoC (System-on-Chip) Conf.", (2015).

[42] Gupta, U., S. Korrapati, N. Matturu and U. Y. Ogras, "A Generic Energy Optimization Framework for Heterogeneous Platforms Using Scaling Models", Microprocessors and Microsystems **40**, 74–87 (2016).

[43] Gupta, U. and U. Ogras, "Constrained Energy Optimization in Heterogeneous Platforms using Generalized Scaling Models", IEEE Comp. Arch. Letters (2014).

[44] Gupta, U. and U. Y. Ogras, "Extending Networks from Chips to Flexible and Stretchable Electronics", in "Proc. of Networks-on-Chip Symp.", (2016).

[45] Gupta, U., J. Park, H. Joshi and U. Y. Ogras, "Flexibility-aware Systems on Polymer: Concept to Prototype", IEEE Trans. on Multi Scale Comput. Sys. **3**, 1, 36–49 (2017).

[46] Gupta, U., C. A. Patil, G. Bhat, P. Mishra and U. Y. Ogras, "DyPO: Dynamic Pareto Optimal Configuration Selection for Heterogeneous MpSoCs", ACM Tran. on Embedded Comp. Sys. (to appear) (2017).

[47] Gupta, U. *et al.*, "Dynamic Power Budgeting for Mobile Systems Running Graphics Workloads", IEEE Trans. on Multi-Scale Comp. Sys. (2017).

[48] Guthaus, M. R., J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge and R. B. Brown, "Mibench: A Free, Commercially Representative Embedded Benchmark Suite", in "Proc. of the Intl. Workshop on Workload Char.", pp. 3–14 (2001).

[49] Hamilton, J. D., *Time Series Analysis*, vol. 2 (Princeton University Press, 1994).

[50] Hanumaiah, V., D. Desai, B. Gaudette, C.-J. Wu and S. Vrudhula, "STEAM: a Smart Temperature and Energy Aware Multicore Controller", Trans. on Embedded Comp. Sys. **13**, 5s, 151 (2014).

[51] Hardkernel, "Platforms, ODROID − XU3", `http://www.hardkernel.com/main/products/prdt_info.php?g_code=G143452239825` (2017).

[52] Henkel, J., H. Khdr, S. Pagani and M. Shafique, "New Trends in Dark Silicon", in "Proc. of the Design Automation Conf.", pp. 1–6 (2015).

[53] Henkel, J. *et al.*, "Dark Silicon: From Computation to Communication", in "Proc. of the Intl. Symp. on Networks-on-Chip", p. 23 (2015).

[54] Herbert, S. and D. Marculescu, "Analysis of Dynamic Voltage/Frequency Scaling in Chip-Multiprocessors", in "Proc. of the Intl. Symp. on Low Power Elec. and Design", pp. 38–43 (2007).

[55] Hoerl, A. E. and R. W. Kennard, "Ridge regression: Biased Estimation for Nonorthogonal Problems", Technometrics, Taylor & Francis Group **12**, 1, 55–67 (1970).

[56] Hot Chips, "Power Management Architecture of the 2nd Generation Intel Core Microarchitecture", http://www.hotchips.org/wp-content/uploads/hc_archives/hc23/HC23.19.9-Desktop-CPUs/HC23.19.921.SandyBridge_Power_10-Rotem-Intel.pdf (2017).

[57] Hsieh, C.-Y., J.-G. Park, N. Dutt and S.-S. Lim, "Memory-Aware Cooperative CPU-GPU DVFS Governor for Mobile Games", in "Proc. of the Symp. on Embedded Sys. For Real-time Multimedia", pp. 1–8 (2015).

[58] Hu, X., Y. Xu, J. Ma, G. Chen, Y. Hu and Y. Xie, "Thermal-sustainable power budgeting for dynamic threading", in "Proc. of the Design Automation Conf.", (2014).

[59] Intel Corp., *Open Source HD Graphics Programmers' Reference Manual* (2015).

[60] Intel Corp., "Intel GPU Tools", http://01.org/linuxgraphics/gfx-docs/igt/ (2016).

[61] Intel Corp., "Minnowboard", http://www.minnowboard.org/ (2016).

[62] Intel Corp, "Atom™ μP $Z3775$", http://ark.intel.com/products/80268 (2017).

[63] Intel Corp., "PRM for Open Source HD Graphics", https://01.org/sites/default/files/documentation/intel-gfx-prm-osrc-skl-vol14-observability.pdf (2017).

[64] Intel Corporation, "Intel© Atom™ Processor $Z2760$", http://ark.intel.com/products/70105 (2017).

[65] Isci, C., G. Contreras and M. Martonosi, "Live, Runtime Phase Monitoring and Prediction on Real Systems With Application to Dynamic Power Management", in "Proc. of the Intl. Symp. on Microarch.", pp. 359–370 (2006).

[66] Isci, C. *et al.*, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget", in "Proc. of Intl. Symp. on Microarch.", (2006).

[67] Ismail, M. and J. Principe, "Equivalence Between RLS Algorithms and the Ridge Regression Technique", in "Proc. of the Conf. on Signals, Sys. and Comp.", pp. 1083–1087 (1996).

[68] James, G., D. Witten, T. Hastie and R. Tibshirani, *An Introduction to Statistical Learning*, vol. 6 (Springer, 2013).

[69] Jin, T., S. He and Y. Liu, "Towards Accurate GPU Power Modeling for Smartphones", in "Proc. of the 2nd Workshop on Mobile Gaming", pp. 7–11 (2015).

[70] Kadjo, D., R. Ayoub, M. Kishinevsky and P. V. Gratz, "A Control-Theoretic Approach for Energy Efficient CPU-GPU Subsystem in Mobile Platforms", in "Proc. of the Design Automation Conf.", pp. 62:1–62:6 (2015).

[71] kernel.org, "Linux Power Capping Framework Documentation", `https://www.kernel.org/doc/Documentation/power/powercap/powercap.txt` (2017).

[72] Khan, M. U. K., M. Shafique and J. Henkel, "Hierarchical Power Budgeting for Dark Silicon Chips", in "Proc. of the Intl. Symp. on Low Power Electronics and Design (ISLPED)", pp. 213–218 (2015).

[73] Kim, R. G. *et al.*, "Wireless NoC and Dynamic VFI Codesign: Energy Efficiency Without Performance Penalty", IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **24**, 7, 2488–2501 (2016).

[74] Kim, R. G. *et al.*, "Imitation learning for dynamic vfi control in large-scale manycore systems", IEEE Trans. on Very Large Scale Integration Syst. **25**, 9, 2458–2471 (2017).

[75] Kreisselmeier, G., "Stabilized Least-Squares Type Adaptive Identifiers", Tran. on Automatic Control **35**, 3, 306–310 (1990).

[76] Kultursay, E., K. Swaminathan, V. Saripalli, V. Narayanan, M. T. Kandemir and S. Datta, "Performance Enhancement Under Power Constraints Using Heterogeneous Cmos-TFET Multicores", in "Proc. of the Intl. Conf. on Hardware/Software Codesign and Sys. Synth.", pp. 245–254 (2012).

[77] Lattner, C., "LLVM and Clang: Next Generation Compiler Technology", in "Proc. of the BSD", pp. 1–2 (2008).

[78] Lattner, C. and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", in "Proc. of the Intl. Symp. on Code Gen. and Opt.: Feedback-directed and Runtime Opt.", p. 75 (2004).

[79] Lee, J. and N. S. Kim, "Optimizing Throughput of Power-and Thermal-Constrained Multicore Processors Using DVFS and Per-Core Power-Gating", in "Prof. of the Design Autom Conf.", pp. 47–50 (2009).

[80] Li, J. and J. F. Martinez, "Dynamic Power-Performance Adaptation of Parallel Computation on Chip Multiprocessors", in "Proc. of the Intl. Symp. on High-Perf. Comp. Arch.", pp. 77–87 (2006).

[81] Lim, H., A. Kansal and J. Liu, "Power Budgeting for Virtualized Data Centers", in "Proc. of the USENIX Annual Tech. Conf.", p. 59 (2011).

[82] Ljung, L., "Characterization of the Concept of 'Persistently Exciting' in the Frequency Domain", Report TFRT **3038** (1971).

[83] Ma, K., X. Wang and Y. Wang, "DPPC: Dynamic Power Partitioning and Control for Improved Chip Multiprocessor Performance", IEEE Trans. on Comp. **63**, 7, 1736–1750 (2014).

[84] Mathworks, "MATLAB System Identification Toolbox", https://www.mathworks.com/products/sysid.html (2017).

[85] Mendel, J. M., *Lessons in Estimation Theory for Signal Processing, Communications, and Control* (Pearson Educ., 1995).

[86] Mercati, P., R. Ayoub, M. Kishinevsky, E. Samson, M. Beuchat, F. Paterna and T. Š. Rosing, "Multi-variable Dynamic Power Management for the GPU Subsystem", in "Proc. of the Design Autom. Conf.", p. 2 (2017).

[87] Milek, J., *Stabilized Adaptive Forgetting in Recursive Parameter Estimation*, no. 4 (vdf Hochschulverlag AG, 1995).

[88] Mishra, Nikita and Zhang, Huazhe and Lafferty, John D and Hoffmann, Henry, "A Probabilistic Graphical Model-Based Approach for Minimizing Energy Under Performance Constraints", in "ACM SIGARCH Computer Architecture News", vol. 43, pp. 267–281 (2015).

[89] Mochel, P., "The sysfs Filesystem", in "Linux Symposium", p. 313 (2005).

[90] Mochocki, B., K. Lahiri and S. Cadambi, "Power Analysis of Mobile 3D Graphics", in "Proc. of the Design, Autom. and Test in Europe Conf.", pp. 502–507 (2006).

[91] Motorola, "Moto X Pure Edition Smartphone", `https://www.motorola.com/us/products/moto-x-pure-edition` (2017).

[92] Mucci, P. J., S. Browne, C. Deane and G. Ho, "PAPI: A Portable Interface to Hardware Performance Counters", in "Proc. of the Department of Defense HPCMP Users Group Conf.", (1999).

[93] Mudge, T., "Power: A First-class Architectural Design Constraint", Computer **34**, 4, 52–58 (2001).

[94] Muthukaruppan, T. S., M. Pricopi, V. Venkataramani, T. Mitra and S. Vishin, "Hierarchical Power Management for Asymmetric Multi-Core in Dark Silicon Era", in "Proc. of the Design Autom. Conf.", pp. 1–9 (2013).

[95] Muztoba, M., U. Gupta, M. Tanvir and U. Y. Ogras, "Robust Communication with IoT Devices using Wearable Brain Machine Interfaces", in "Proc. of Int. Conf. on Computer-Aided Design", (2015).

[96] Nagasaka, H. *et al.*, "Statistical Power Modeling of GPU Kernels using Performance Counters", in "Proc. of the Intl. Green Comp. Conf.", pp. 115–122 (2010).

[97] National Instr., "NI USB-6289", `http://sine.ni.com/nips/cds/view/p/lang/en/nid/209154` (2015).

[98] ODROID, " XU+E Platform", `http://www.hardkernel.com/` (2017).

[99] Ogras, U. Y., R. Z. Ayoub, M. Kishinevsky and D. Kadjo, "Managing Mobile Platform Power", in "Proc. of Intl. Conf. on Computer-Aided Design", pp. 161–162 (2013).

[100] Ogras, U. Y. and R. Marculescu, *Modeling, Analysis and Optimization of Network-on-Chip Communication Architectures*, vol. 184 (Springer Science & Business Media, 2013).

[101] Ogras, U. Y., R. Marculescu, D. Marculescu and E. G. Jung, "Design and Management of Voltage-Frequency Island Partitioned Networks-on-Chip", IEEE Trans. on Very Large Scale Integration Systems **17**, 3, 330–341 (2009).

[102] Pagani, S. *et al.*, "TSP: Thermal Safe Power: Efficient Power Budgeting for Many-Core Systems in Dark Silicon", in "Proc. of the Intl. Conf. on Hardware/Software Codesign and System Synthesis", p. 10 (2014).

[103] Palermo, G., C. Silvano and V. Zaccaria, "Multi-objective Design Space Exploration of Embedded Systems", Jrnl of Embd. Comp. **1.3**, 305–316 (2005).

[104] Palesi, M. and T. Givargis, "Multi-objective Design Space Exploration Using Genetic Algorithms", in "Proc. of the Intl. Symp. on Hardware/Software Codesign", pp. 67–72 (2002).

[105] Pallipadi, V., S. Li and A. Belay, "Cpuidle: Do Nothing, Efficiently", in "Proc. of the Linux Symp.", vol. 2, pp. 119–125 (2007).

[106] Pallipadi, V. and A. Starikovskiy, "The Ondemand Governor", in "Proc. of the Linux Symp.", vol. 2 (2006).

[107] Panda, P. R., B. Silpa, A. Shrivastava and K. Gummidipudi, *Power-efficient System Design* (Springer Science & Business Media, 2010).

[108] Park, J.-G., C.-Y. Hsieh, N. Dutt and S.-S. Lim, "Co-Cap: Energy-Efficient Cooperative CPU-GPU Frequency Capping for Mobile Games", in "Proc. of the ACM Symp. on Applied Computing", pp. 1717–1723 (2016).

[109] Paterna, F., U. Gupta, R. Ayoub, U. Y. Ogras and M. Kishinevsky, "Adaptive Performance Sensitivity Model to Support GPU Power Management", in "Proc. of the Workshop on Autotuning and Adaptivity Approaches for Energy Efficient HPC Sys.", p. 5 (2017).

[110] Pathak, A., Y. C. Hu, M. Zhang, P. Bahl and Y.-M. Wang, "Fine-Grained Power Modeling for Smartphones using System Call Tracing", in "Proc. of the ACM Conf. on Computer systems", pp. 153–168 (2011).

[111] Pathania, A., A. E. Irimiea, A. Prakash and T. Mitra, "Power-Performance Modelling of Mobile Gaming Workloads on Heterogeneous MPSoCs", in "Proc. of the Design Autom. Conf.", pp. 201:1–201:6 (2015).

[112] Petoumenos, P., L. Mukhanov, Z. Wang, H. Leather and D. S. Nikolopoulos, "Power Capping: What Works, What Does Not", in "Proc. of the Intl. Conf. on Parallel and Distributed Systems", pp. 525–534 (2015).

[113] Pothukuchi, R. P., A. Ansari, P. Voulgaris and J. Torrellas, "Using Multiple Input, Multiple Output Formal Control to Maximize Resource Efficiency in Architectures", in "Proc. of the Intl. Symp. on Computer Architecture (ISCA)", pp. 658–670 (2016).

[114] Power, J., J. Hestness, M. Orr, M. Hill and D. Wood, "gem5-gpu: A Heterogeneous CPU-GPU Simulator", IEEE Comp. Arch. Letters (2014).

[115] Prakash, A., H. Amrouch, M. Shafique, T. Mitra and J. Henkel, "Improving Mobile Gaming Performance Through Cooperative CPU-GPU Thermal Management", in "Proc. of the Design Automation Conf.", p. 47 (2016).

[116] Qualcomm Inc., "Trepn profiler", `https://developer.qualcomm.com/software/trepn-power-profiler` (2017).

[117] Rabaey, J. M., M. Pedram *et al.*, *Low Power Design Methodologies*, vol. 118 (Kluwer Academic Publishers Norwell, 1996).

[118] Raghavan, A., Y. Luo, A. Chandawalla, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch and M. M. Martin, "Computational Sprinting", in "Proc. of Symp. on High Performance Computer Architecture", (2012).

[119] Reda, S., R. Cochran and A. K. Coskun, "Adaptive Power Capping for Servers with Multithreaded Workloads", IEEE Micro **5**, 32, 64–75 (2012).

[120] Ren, Z., B. H. Krogh and R. Marculescu, "Hierarchical Adaptive Dynamic Power Management", IEEE Trans. on Computers **54**, 4, 409–420 (2005).

[121] Rotem, E., A. Naveh, A. Ananthakrishnan, E. Weissmann and D. Rajwan, "Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge", IEEE Micro **32**, 2, 20–27 (2012).

[122] Sahin, O., P. T. Varghese and A. K. Coskun, "Just Enough is More: Achieving Sustainable Performance in Mobile Devices under Thermal Limitations", in "Proc. of the Intl. Conf. on Computer-Aided Design", pp. 839–846 (2015).

[123] Sayed, A. H., *Fundamentals of Adaptive Filtering* (John Wiley & Sons, 2003).

[124] Sayed, A. H., *Adaptive Filters* (John Wiley & Sons, 2011).

[125] Shafique, M., S. Garg, J. Henkel and D. Marculescu, "The EDA Challenges in the Dark Silicon Era", in "Proc. of the Design Automation Conf.", pp. 1–6 (2014).

[126] Sherwood, T., E. Perelman, G. Hamerly, S. Sair and B. Calder, "Discovering and Exploiting Program Phases", IEEE micro **23**, 6, 84–93 (2003).

[127] Singh, A. K. and B. Bhadauria, "Finite Difference Formulae for Unequal Subintervals using Lagranges Interpolation Formula", Int. J. Math. Anal **3**, 17, 815 (2009).

[128] Singla, G., G. Kaur, A. K. Unver and U. Y. Ogras, "Predictive Dynamic Thermal and Power Management for Heterogeneous Mobile Platforms", in "Proc. of the Conf. on Design, Automation & Test in Europe", pp. 960–965 (2015).

[129] Skadron, K., M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan and D. Tarjan, "Temperature-Aware Computer Systems: Opportunities and Challenges", IEEE Micro **23**, 6, 52–61 (2003).

[130] Skadron, K., M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy and D. Tarjan, "Temperature-Aware Microarchitecture: Modeling and Implementation", ACM Tran. on Arch. and Code Optimization **1**, 1, 94–125 (2004).

[131] Song, T., D. Lo and G. E. Suh, "Prediction-Guided Performance-Energy Tradeoff with Continuous Run-Time Adaptation", in "Proc. of the Intl. Symp. on Low Power Elec. and Design", pp. 224–229 (2016).

[132] Souza, F. and R. Araújo, "An Online Variable Selection Method Using Recursive Least Squares", in "Proc. of the Emerging Tech. & Factory Autom.", pp. 1–8 (2012).

[133] Statista, "Number of Mobile Phone Users Worldwide From 2013 to 2019", `https://www.statista.com/statistics/274774/forecast-of-mobile-phone-users-worldwide/` (2017).

[134] Statista, "Worldwide mobile app revenues in 2015, 2016 and 2020", `https://www.statista.com/statistics/269025/worldwide-mobile-app-revenue-forecast/` (2017).

[135] Strang, G., *Computational Science and Engineering*, vol. 791 (Wellesley-Cambridge Press, 2007).

[136] Su, B. *et al.*, "PPEP: Online Performance, Power, and Energy Prediction Framework and DVFS Space Exploration", in "Proc. of the Intl. Symp. on Microarch.", pp. 445–457 (2014).

[137] Techreport, "BayTrail Arch.", `http://techreport.com/review/25329/intel-atom-z3000-bay-trail-soc-revealed` (2016).

[138] Thomas, S. *et al.*, "CortexSuite: A Synthetic Brain Benchmark Suite", in "Proc. of the Intl. Symp. on Workload Char.", pp. 76–79 (2014).

[139] TI-INA231, `http://www.ti.com/lit/ds/symlink/ina231.pdf` (2017).

[140] Turner, C. S., "A Fast Binary Logarithm Algorithm", Streamlining Digital Signal Processing: A Tricks of the Trade Guidebook pp. 281–283 (2012).

[141] Vallina-Rodriguez, N. and J. Crowcroft, "Energy Management Techniques in Modern Mobile Handsets", IEEE Comm. Surveys & Tutorials , 99, 1–20 (2012).

[142] Varatkar, G. V. and R. Marculescu, "On-chip Traffic Modeling and Synthesis for MPEG-2 Video Applications", IEEE Trans. on Very Large Scale Integration Systems **12**, 1, 108–119 (2004).

[143] Wang, H., V. Sathish, R. Singh, M. J. Schulte and N. S. Kim, "Workload and power budget partitioning for single-chip heterogeneous processors", in "Proc. of Parallel Arch. and Compilation", (2012).

[144] Wang, W., P. Mishr A and S. Ranka, *Dynamic Reconfiguration in Real-Time Systems* (Springer, 2012).

[145] Wang, X., K. Ma and Y. Wang, "Adaptive Power Control With Online Model Estimation for Chip Multiprocessors", IEEE Trans. on Parallel and Distributed Sys. **22**, 10, 1681–1696 (2011).

[146] Wang, X. *et al.*, "A Pareto-Optimal Runtime Power Budgeting Scheme for Many-Core Systems", Microprocessors and Microsystems **46**, 136–148 (2016).

[147] XDA-Developers Forums, `https://forum.xda-developers.com/general/general/ref-to-date-guide-cpu-governors-o-t3048957` (2017).

[148] Zakharov, Y. V., G. P. White and J. Liu, "Low-complexity RLS Algorithms Using Dichotomous Coordinate Descent Iterations", IEEE Tran. on Signal Proc. **56**, 7, 3150–3161 (2008).

[149] Zhan, X. and S. Reda, "Techniques For Energy-Efficient Power Budgeting In Data Centers", in "Proc. of the Design Automation Conf.", p. 176 (2013).

[150] Zhang, H. and H. Hoffmann, "Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques", in "Proc. of the Intl. Conf. on Arch. Support for Programming Languages and Operating Systems", pp. 545–559 (2016).

[151] Zheng, X., L. K. John and A. Gerstlauer, "Accurate Phase-level Cross-platform Power and Performance Estimation", in "Proc. of Design Autom. Conf.", p. 4 (2016).

[152] Zhu, Y. and V. J. Reddi, "High-Performance and Energy-Efficient Mobile Web Browsing on Big/Little Systems", in "Intl. Symp. on High Perf. Comput. Arch.", (2013).

[153] Zhuo, J. and C. Chakrabarti, "Energy-Efficient Dynamic Task Scheduling Algorithms for DVS Systems", ACM Trans. on Embedded Computing Systems **7**, 2, 17 (2008).

VITA

EDUCATION

Ph.D., Electrical Engineering *08/2014 - 05/2018*

Arizona State University, Tempe, Arizona, USA

M.S., Electrical Engineering *08/2012 - 08/2014*

Arizona State University, Tempe, Arizona, USA

B.E., Electronics & Communication Engineering *08/2007 - 11/2011*

Manipal University, Manipal, Karnataka, India