UNIVERSITY OF CALIFORNIA,
IRVINE

Cooperative CPU-GPU Dynamic Power Management Methodologies for Energy-efficient Mobile Gaming

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Jurn-Gyu Park

Dissertation Committee:
Professor Nikil Dutt, Chair
Professor Alex Nicolau
Professor Mohammad Al Faruque

2017

DEDICATION

To my God

The LORD is my shepherd, I shall not be in want.
He makes me lie down in green pastures, he leads me beside quiet waters,
he restores my soul. He guides me in paths of righteousness for his name's sake.
Even though I walk through the valley of the shadow of death, I will fear no evil,
for you are with me; your rod and your staff, they comfort me.
You prepare a table before me in the presence of my enemies.
You anoint my head with oil; my cup overflows.
Surely goodness and love will follow me all the days of my life, and
I will dwell in the house of the LORD forever.

- Psalms 23 -

To my Wife

A wife of noble character who can find?
She is worth far more than rubies.

- Proverbs 31:10 -

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Above all, I would like to give thanks to my God who has been giving me knowledge, wisdom, endurance and strength to finish this long journey.

Foremost, I truly would like to thank my advisor, Professor Nikil Dutt for his great generosity and excellence in his supervision. He gave me huge freedom and infinite encouragement to explore my ideas, and taught me how to discover systematic methodologies for research problems with logical thinking and how to organize and present our work in literature and talk. The time that I have spent with him itself was blessing and learning for me.

Sincerely, I would also like to thank Professor Alex Nicolau and Mohammad Al Faruque for their guidance, advice and serving on my thesis committee. Especially, the comments during the candidacy exam and the topic exam were good guidelines and crucial checkpoints for my next steps.

I truly give thanks to Professor Sung-soo Lim who gave me a lot of inspiration with logical and sharp comments for our works. I will forever be grateful for the time and the efforts he has spent in my studies. I also want to thank KIAT, Ministry of Trade, Industry and Energy, South Korea for providing Global Research Collaboration Project, UC Irvine's graduate division and Melanie Sander and Grace Wu at CECS for their help.

Another blessing during my PhD studies is the people whom I met through Dutt's research group from my labmates to visiting scholars. Luis (Danny) Bathen, Kazuyuki Tanimura, Jun Young Shin, Abbas Banaiyan, Santanu Sarma and Hossein Tajik already became doctors; and Majid Namaki Shoushtari, Bryan (Donny) Donyanavard, Roger Chen-Ying Hsieh, Tiago Rogerio Muck, Kasra Moazzemi, Hamid Nejatollahi and Hirak Kashyap are with me now. In addition, I would like to thank the visiting scholars: Professors Gu-Min Jeong, Hoyoung Hwang, Yukio Mitsuyama, Alfonso Avila, Hiroyuki Tomiyama, Antonio Augusto Frohlich, MyungKeun Yoon, Amir Rahmani and Bruno Zatt, and Dr. Janmartin Jahn, Dr. Trent Lo, Dr. Gustavo Girao, Juan Gonzalez, Amir Mahdi Monazzah, Andre Martins. Especially, Professor Kanghee Kim and other Korean professors provided me heartfelt encouragement and practical advice with refreshments thankfully, and Kookmin I-SURF interns (esp. Hoyeonjiki Kim and Hyungjun Lee) gave me a lot of help practically in my research.

I give thanks to my friends. Jiman Ok, Junkyu Lee, Taewoo Kim and their families came together from our home country to UCI in the same year, and we were very supportive of each other. And, I would like to thank Disciple Community Church, pastor Hyunjong Ko, my small group leaders (Hyeokseoung Park and Sanghoon Jun), small group members, discipleship training members, Ohana teachers and my students. They will be my eternal fellow workers spiritually.

Lastly, this thesis would not have been completed without the support and patience of my faithful wife, Yeonji Lee. She has been preparing my lunch box for five years with full of her love, and has been encouraging me always with love and trust. Moreover, my three children, Heechan and Heeon and Eunchan, are always my joy, strength and power to overcome every challenging circumstance. This thesis is a fruit of infinite love, sacrifices and prayers from my family, my parents and my wife's parents.

CURRICULUM VITAE

Jurn-Gyu Park

EDUCATION

Doctor of Philosophy in Computer Science **2017**
University of California, Irvine *Irvine, CA, USA*

Master of Engineering in Computer Engineering **2012**
Yonsei University *Seoul, Korea*

Bachelor of Science in Mechanical and Automotive Engineering **2001**
Kookmin University *Seoul, Korea*

WORK and RESEARCH EXPERIENCE

Graduate Research Assistant **2012–2017**
University of California, Irvine *Irvine, CA, USA*

Software Engineer/Technical Lecturer **2006–2011**
MDS Technology *Seoul, Korea*

Planning/Marketing/Sales Engineer **2002–2005**
MDS Technology *Seoul, Korea*

TEACHING EXPERIENCE

Teaching Assistant/Reader **2012–2017**
University of California, Irvine *Irvine, CA, USA*

Technical Lecturer **2006–2011**
MDS Technology *Seoul, Korea*

REFEREED CONFERENCE PUBLICATIONS

HiCAP: Hierarchical FSM-based Dynamic Integrated CPU-GPU Frequency Capping Governor for Energy-Efficient Mobile Gaming — **Aug. 2016**
International Symposium on Low Power Electronics and Design (ISLPED)

Co-Cap: Energy-efficient Cooperative CPU-GPU Frequency Capping for Mobile Games — **Apr. 2016**
Symposium on Applied Computing (SAC)

Memory-aware Cooperative CPU-GPU DVFS Governor for Mobile Games — **Oct. 2015**
Symposium on Embedded Systems for Real Time Multimedia (ESTIMedia)

Quality-aware Mobile Graphics Workload Characterization for Energy-efficient DVFS Design — **Oct. 2014**
Symposium on Embedded Systems for Real Time Multimedia (ESTIMedia)

TECHNICAL REPORTS

Using HSFMs to Model Mobile Gaming Behavior for Energy Efficient DVFS Governors — **Jun. 2016**
UC Irvine, Center for Embedded and Cyber-physical Systems, CECS TR 16-02

Cooperative CPU-GPU Frequency Capping (Co-Cap) for Energy Efficient Mobile Gaming — **Dec. 2015**
UC Irvine, Center for Embedded and Cyber-physical Systems, CECS TR 15-05

ABSTRACT OF THE DISSERTATION

Cooperative CPU-GPU Dynamic Power Management Methodologies
for Energy-efficient Mobile Gaming

By

Jurn-Gyu Park

Doctor of Philosophy in Computer Science

University of California, Irvine, 2017

Professor Nikil Dutt, Chair

One of the fundamental challenges to contemporary mobile platforms deploying heterogeneous CPU-GPU based architectures that execute mobile games and other graphics-intensive applications is to design software governors through Dynamic Voltage Frequency Scaling (DVFS) for achieving high performance with energy-efficiency on battery-based systems. However, separate CPU and GPU governors miss opportunities for further energy savings through cooperative/integrated CPU-GPU power management. Contemporary integrated CPU-GPU governors for diverse and dynamic gaming workloads utilize classical statistical or heuristic models with a small set of mobile games for both modeling and evaluation resulting in high prediction errors with lost potential for energy savings. To overcome these limitations, this thesis presents a comprehensive graphics workload characterization by developing custom micro-benchmarks and then proposes three different cooperative CPU-GPU dynamic power management methodologies by using large sets of real games and micro-benchmarks. As a first step, we present a study of mobile GPU graphics workload characterization for DVFS design considering performance and energy efficiency on a real smart-phone. We develop micro-benchmarks that stress specific stages of the graphics pipeline separately, and analyze the relationship between varying graphics workloads and resulting energy and performance of different mobile graphics pipeline stages. We then propose a simple yet effective

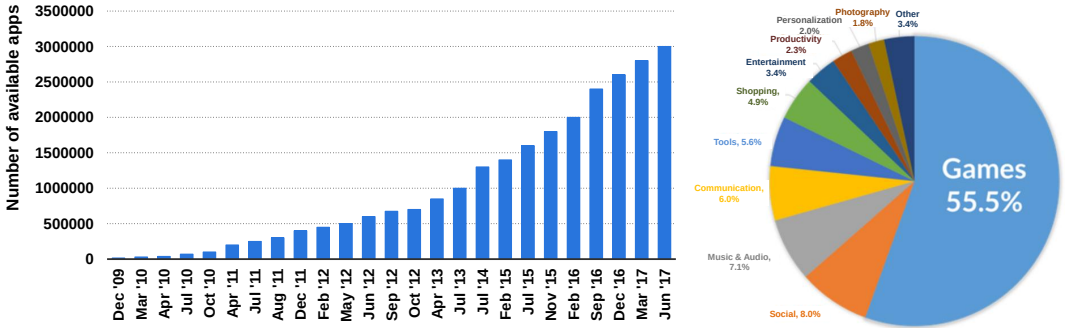
strategy called Co-Cap (Cooperative Frequency-Capping), a cooperative CPU-GPU DVFS strategy that orchestrates energy-efficient CPU and GPU DVFS through coordinated CPU and GPU frequency capping to avoid frequency over-provisioning while maintaining desired performance. Furthermore, we propose a model-based DVFS design approach, a Hierarchical Finite State Machine (HFSM) based CPU-GPU governor that models the dynamic behavior of mobile gaming workloads, and applies a cooperative, dynamic CPU-GPU frequency-capping policy to yield energy efficiency adapting to the games' inherent dynamism. Finally, we present a machine learning enhanced integrated CPU-GPU governor that builds tree-based piecewise linear prediction models resulting in high accuracy and low complexity of cost functions using practical offline machine learning techniques, and integrate the models for online estimation into an integrated CPU-GPU DVFS governor applying piecewise policies based on the models. We demonstrate efficacy of our methodologies across over 100 real games and a few hundred custom micro-benchmarks, achieving substantial energy efficiency gains of up to 18% improvement in energy-per-frame over existing governor policies, with minimal loss in performance.

Chapter 1

Introduction

1.1 Smartphone Systems

A smartphone system is a mobile personal computer with a mobile operating system designed for typical mobile or handheld usage scenarios [96]. With the emergence of Apple's iPhone [11] (2007) (one of the first smartphones to use a multi-touch interface) and the first phone using Android [32] (2008) (an open-source platform owned by Google), smartphones started to gain widespread popularity with the help of application stores such as Apple's



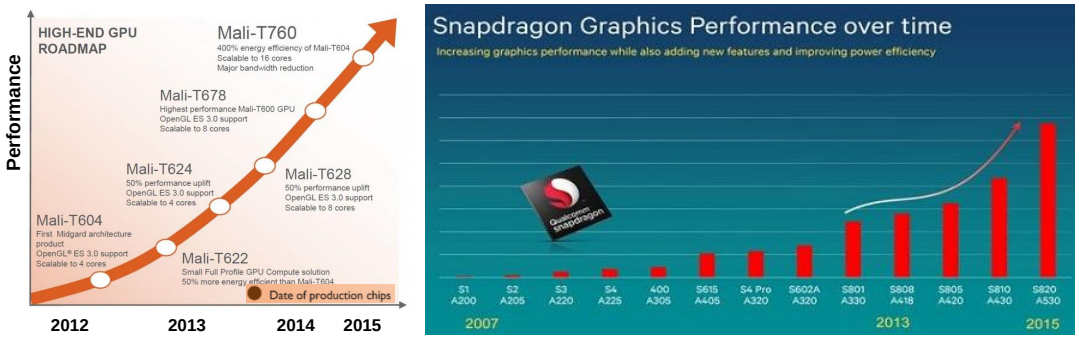
(a) Number of Apps in the Google Play Store [92]

(b) US Google Play Top Free Chart by Category [71]

Figure 1.1: Trends of Smartphone Applications

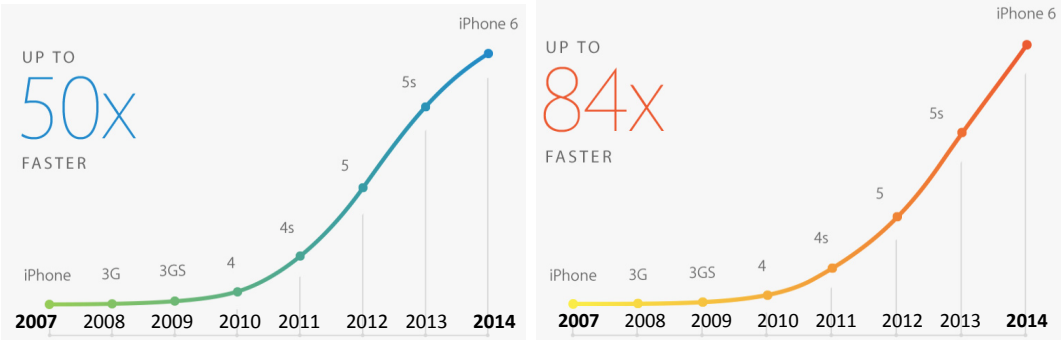
App Store [10] and Google's Android Market (now Google Play Store [33]). For example, as shown in Figure 1.1, the number of available applications in these application stores is increasing dramatically, with games dominating as one of the most popular application category.

The increasing use of graphics-intensive mobile games and other applications has necessitated the deployment of high-performance Heterogeneous MultiProcessor Systems-on-Chip (HMPSoC) with integrated GPUs to deliver desired performance. Figure 1.2.(a) shows the ARM Mali high-end GPU processor roadmap, highlighting the GPU performance increasing rapidly. This pattern is almost similar to the roadmap of Qualcomm GPU processor (Figure 1.2.(b)); tracking changes in the GPU porcessors, we note that the Snapdragon graphics performance increases dramatically after 2013.



(a) ARM GPU Performance [12] (b) Qualcomm Snapdragon GPU Performance [56]

Figure 1.2: Change of High-performance Mobile Integrated GPUs



(a) Apple CPU Performance (b) Apple GPU Performance

Figure 1.3: Change of Apple Smartphone Platforms [29]

A similar trend can be observed with Apple's smartphone platforms (Figure 1.3): after

the first emergence of iPhone in 2007, only 7 years later the iPhone 6 platform shows a 50x increase in CPU performance and 84x increase in GPU performance.

However, high performance mobile HMPSoCs result in high power consumption in the CPU and GPU. Figure 1.4 shows the correlation between performance and power in mobile

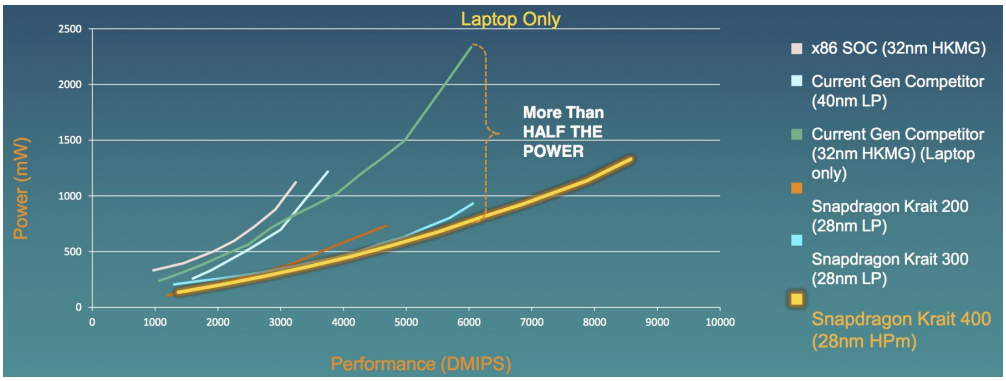


Figure 1.4: Correlation between Performance and Power [7] (The performance metric is Drystone MIPS and power consumption unit is mW)

SoCs. Even though the performance range and the slope (power-efficiency) are different for each processor, clearly the correlation between performance and power is proportional across all processors. Moreover, Figure 1.5 shows the comparison of power consumption between CPU and GPU using a graphics-intensive gaming benchmark, the GL Egypt benchmark. Here we note that at the beginning, the average GPU power consumption is much lower

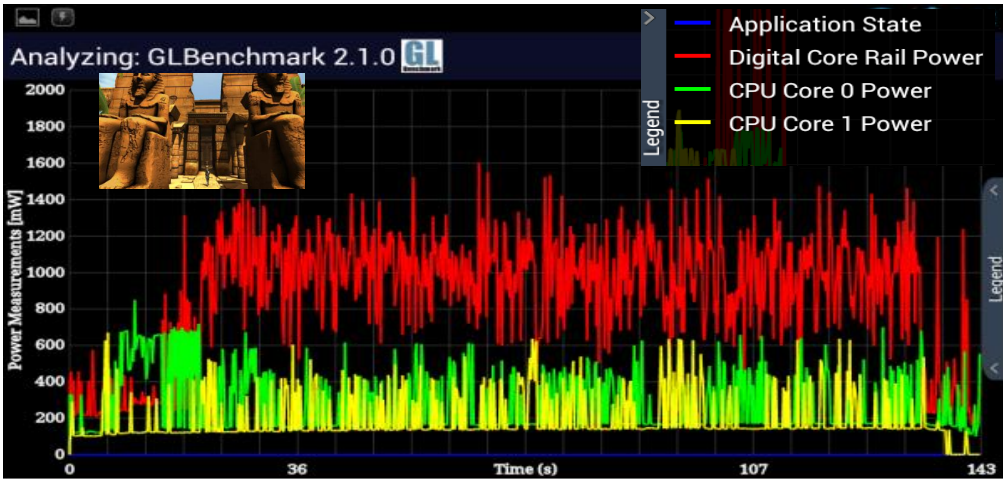


Figure 1.5: Comparison of CPU and GPU power consumption [5]: x-axis: time in second, y-axis: Power (mW)

than the power consumption of CPUs because the application loading phase comes before

the regular rendering phase; then GPU power consumption is overwhelmingly higher than CPU power because the benchmark is a GPU graphics-intensive application. In addition, the CPU and GPU power consumptions are also changing dynamically with high variation between maximum and minimum power.

In summary, the trends of high performance heterogeneous mobile platforms with graphics-intensive mobile games and other applications result in high CPU and GPU power consumption with high variation (dynamism).

1.2 Motivation and Challenges

Contemporary mobile platforms use software governors that deploy Dynamic Voltage Frequency Scaling (DVFS) techniques to achieve high performance with energy-efficiency for heterogeneous CPU-GPU based architectures executing mobile games and other graphics-intensive applications.

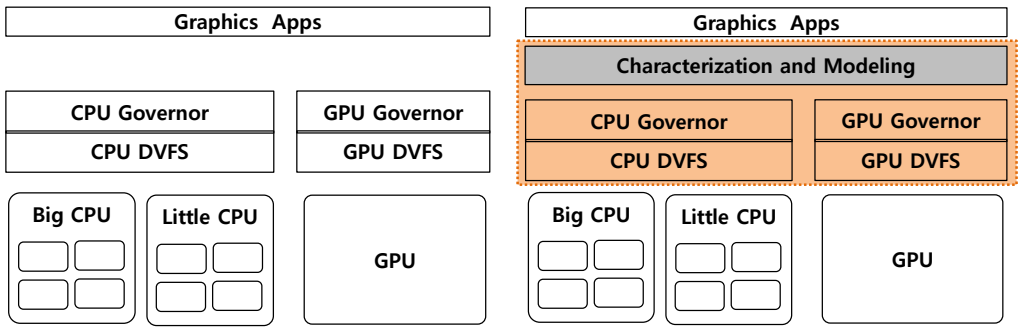
After a comprehensive study of CPU and GPU DVFS for gaming workloads on mobile heterogeneous platforms, we make the following observations: 1) There have been no previous systematic studies to correlate the performance, power, and energy efficiency of mobile GPUs based on diverse graphics workloads to enable more efficient mobile platform DVFS policies for energy savings. 2) Traditionally, separate CPU and GPU governors are deployed in order to achieve energy efficiency through DVFS, but miss opportunities for further energy savings through coordinated system-level application of DVFS. 3) Mobile games typically exhibit inherent behavioral dynamism, which existing governor policies are unable to exploit effectively to manage CPU/GPU DVFS policies. 4) For dynamic and diverse gaming workloads, existing governors utilize statistical or heuristic models with a small set of mobile games for both modeling and evaluation, resulting in high prediction errors in modeling, and do not exploit practical machine learning approaches for prediction models with high

accuracy and low complexity.

We also note the following challenges. First, for battery-based commercial mobile platforms, performance and power issues should be considered simultaneously to enable system design for two common optimizations: performance optimization under power/energy constraints, or power/energy optimization under performance constraints. Second, mobile graphics workloads (especially gaming workloads) are highly diverse and dynamic, thereby requiring comprehensive graphics workloads characterization for dynamic power management design. Third, mobile platforms are changing rapidly, requiring a simple and easily portable methodology for porting and implementation of DVFS governor policies. Fourth, existing governor policies are typically heuristic based, and could benefit from a model-based methodology to enhance rigor in the design process.

1.2.1 Existing Approaches

Current approaches address these challenges typically in a disjoint manner. The traditional



(a) Separate Governors (b) Cooperative/Integrated Governors

Figure 1.6: State-of-the-art Approaches

approach uses separate CPU and GPU governors (Figure 1.6(a)), that scale voltage and frequency dynamically, but independently for the CPU and GPU components. The CPU governor is in charge of CPU part and vice versa for GPU part, but this approach does not consider any cooperation between CPU and GPU. Some integrated governors [80] [51]

(Figure 1.6(b)) were proposed quite recently. However, they used a small set of specific benchmarks, which result in high prediction errors for unseen workloads that are typical for most mobile applications. Our approach uses a large set of games (over 100 real games and a few hundred custom micro-benchmarks [75]) and uses an automatic measurement tool [79] to generate better experimental data that enables more thorough analysis; and we present several integrated CPU-GPU governor policies to enable synergistic management of mobile platform resources.

1.3 Thesis Overview

The thesis proposes cooperative CPU and GPU dynamic power management (DPM) methodologies for energy-efficient mobile gaming based on comprehensive graphics workload characterization.

As outlined in Figure 1.7, first we present a comprehensive characterization of graphics workloads by design and implementation of gaming micro-benchmarks [75]. We then propose three different cooperative DPM methodologies: 1) A simple but effective lookup-table based frequency-capping methodology [77], 2) A hierarchical-FSM based dynamic behavior modeling methodology for gaming workloads using the frequency-capping technique [78], and 3) A machine-learning enhanced prediction modeling methodology based integrated CPU-GPU DVFS.

1.3.1 Graphics Workload Characterization (GWC)

We observed that graphics workloads exercise different graphics pipeline stages such as the Geometry and Fragment units that greatly affect the performance and the power consumption of the mobile platform. Unfortunately most real-games have mixed various workloads

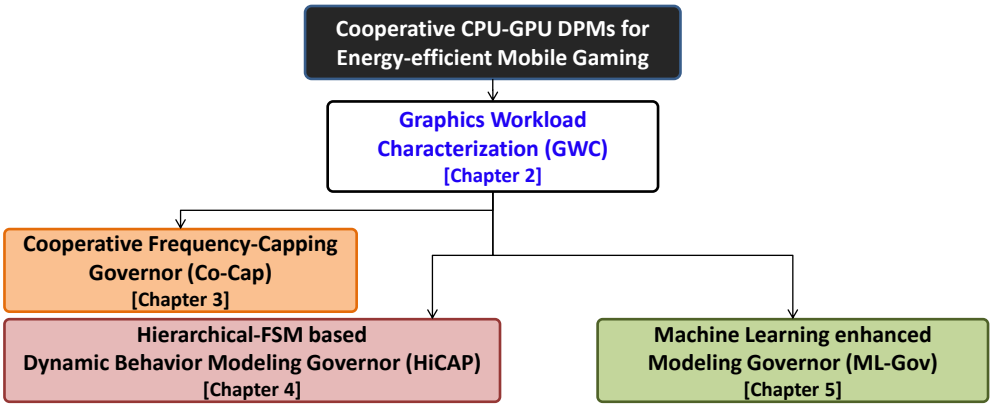


Figure 1.7: Thesis Overview

and release only binary files that prevent modification of their workloads to enable experimental evaluation. Therefore, we propose the design and implementation of graphics micro-benchmarks by stressing different components of the graphics pipeline stages. We focus on the workloads of the three categories (GPU memory, GPU computation and CPU workloads) and design five micro-benchmarks by stressing each specific pipeline stage with fixed workloads in other pipeline stages. For example, for the vertex memory micro-benchmark, we mainly changed the number of vertices; and for the texture memory micro-benchmark, we changed the texture image size. Using these five micro-benchmarks, we studied the impact of workload components on Frames per Second (FPS), power and energy per frame. These micro-benchmarks and the profiled data were then used extensively for workload characterization, workload analysis, model building (learning) and evaluations. We describe our Graphics Workload Characterization (GWC) approach in Chapter 2.

1.3.2 Cooperative Frequency-Capping Governor (Co-Cap)

Recall our motivating observation that commercial platforms (Figure 1.6(a)) have separate CPU and GPU governors with over-provisioned frequencies. For example, commercial CPU governors such as ONDEMAND or INTERACTIVE immediately scale up to the maximum frequency for high performance at a certain threshold; we observe that significant power

reduction with negligible performance degradation can be realized using a slightly lower maximum frequency, called the saturated frequency. However, the challenge is that the saturated frequency depends on different CPU and GPU workloads. Therefore, a large set of diverse gaming benchmarks using various different types of workloads are needed for building a model. Furthermore, mobile platforms are changing rapidly with different computing capability. In order to overcome these challenges, we propose Co-Cap, a cooperative CPU-GPU saturated frequency Lookup table based governor. Co-Cap's a simple and easily portable methodology is applicable for rapidly changing platforms. We describe Co-Cap in Chapter 3.

1.3.3 Hierarchical-FSM based Dynamic Behavior Modeling Governor (HiCAP)

Another motivating observation is that FPS and CPU/GPU workloads are changing dynamically at runtime, providing additional opportunities for energy savings. However, this raises the challenging problem of how to specify dynamic behaviors for gaming workloads. Therefore, we propose HiCAP, a model-based approach that captures dynamic behavior using a hierarchical Finite State Machine (HFSM) model. HFSMs exploit hierarchically an intuitive behavior modeling methodology for complex embedded systems [31] [69]. Furthermore HFSMs naturally capture the inherent hierarchical model in typical game design models [61] [87]. Therefore, we specify each state hierarchically using Quality of Service (QoS) and CPU/GPU workload dominance metrics. First, we specify two super-states: QoS-meet or QoS-loss. If a current FPS during an epoch of time is higher than a target FPS, the state will be called QoS-meet. If not, it is QoS-loss. Then, using CPU and GPU cost, we observe if the state is CPU-dominant (CPU cost > GPU cost) or GPU-dominant (GPU cost > CPU cost). Finally, according to the hierarchical property, we apply a specific policy in each leaf state (i.e., applying different policies for different types of states). We describe HiCAP in Chapter 4.

1.3.4 Machine Learning enhanced Modeling Governor (ML-Gov)

We observed that classical statistical models such as simple or multiple linear regression models suffer from limitations when executing mobile games with dynamic and diverse workloads on CPU-GPU heterogeneous mobile platforms. For example, simple linear prediction models using small data sets result in high prediction errors for unseen workloads. To overcome this limitation, one practical alternative is to use a Machine Learning (M.L) enhanced model building methodology using large amounts of diverse data. M.L approaches are facilitated by the ability to collect large amounts of data from mobile systems, and the access to tools that enable visualization and analysis. Therefore, using this kind of M.L approach, prediction errors and structure of cost functions can be evaluated usefully. Based on these observations, we propose a tree-structured (piecewise) linear model based integrated governor, using practical M.L techniques. Like general M.L approaches, our proposal is composed of a learning phase and a prediction phase. In the learning phase, we use the training data to build tree-based piecewise linear models with evaluations. Then in the prediction phase, we set CPU and GPU frequencies using the models to maximize energy savings with minimal performance degradation. We present our ML-Gov approach in Chapter 5.

1.4 Thesis Contributions

1.4.1 Design and Implementation of micro-benchmarks

For diverse and dynamic gaming workloads, we design and implement our graphics micro-benchmarks by stressing specific graphics pipeline stages separately for graphics workload characterization. Comprehensive observations and thorough analyses from the results of micro-benchmarks provide the correlation between workloads of hardware pipeline stages

and performance/power effects. We present opportunities for energy-efficient mobile DVFS design based on these analyses.

1.4.2 Simple but highly effective CPU-GPU Frequency Capping

For rapidly changing mobile platforms, we propose Co-Cap, a cooperative frequency capping governor to achieve energy efficiency for a diverse set of mobile games by building simple and easily portable CPU-GPU Lookup Tables. Our Co-Cap capping strategy avoids unnecessarily higher frequency of CPU and GPU considering both FPS and per-frame energy saving on top of the default CPU and GPU governors; we present characterization of diverse mobile graphics gaming workloads using combinations of our custom micro-benchmarks to enable efficient dynamic application of frequency capping. We demonstrate the efficacy of Co-Cap across over 100 combined micro-benchmarks and 40 real mobile graphics applications, using multiple training and deployment sets achieving significant improvements in energy efficiency with minimal loss in performance.

1.4.3 Hierarchical FSM-based Dynamic Behavior Modeling

For effective adaptation of dynamic behavior changes, we propose a Hierarchical FSM (HFSM) based dynamic behavior modeling strategy for mobile gaming. We present a cooperative CPU-GPU governor that deploys a simple maximum frequency-capping methodology exploiting the HFSM for dynamic DVFS. We present experimental results on a large set of real mobile games with dynamic behaviors, showing significant energy savings of in Energy-perFrame (EpF) with minimal loss in FPS performance.

1.4.4 Machine Learning enhanced Simple and Accurate Prediction Models

We develop simple and accurate prediction models for diverse and dynamic gaming workloads on heterogeneous mobile platforms, using machine learning enhanced performance models. We build tree-based piecewise linear regression models using off-line machine learning algorithms built in an existing data mining tool. We present an integrated CPU-GPU DVFS governor that applies piecewise policies using analyses of the models. We present experimental results on training and testing sets of mobile games with various characteristics, showing significant energy savings in Energy-per-Frame (EpF).

Thesis Organization The rest of this thesis is organized as follows: Chapter 2 describes graphics workload characterization by design and implementation of graphics micro-benchmarks in mobile embedded systems. Chapter 3 presents a simple but effective Co-Cap methodology for rapidly changing mobile platforms. Chapter 4 presents dynamic behavior modeling based governor by building hierarchical FSM behavior model for effective adaptation of dynamic behavior changes. Chapter 5 presents a machine learning enhanced tree-based piecewise regression models for simple and accurate prediction models. In Chapter 6, we conclude this thesis and address future directions for this research.

Chapter 2

Graphics Workload Characterization for DVFS Design

2.1 Introduction

The increasing use of mobile platforms for 3D games and other graphics-intensive applications has resulted in deployment of high-performance mobile GPUs. Table 2.1 lists sample state-of-the-art smartphones and their corresponding mobile GPUs, their maximum GPU operation frequencies, and the number of Dynamic Voltage Frequency Scaling (DVFS) steps to enable modulation of the mobile GPU energy consumption. This table clearly shows the trend towards higher mobile GPU frequencies (e.g., 400-578 Mhz) in the face of increased graphics-intensive mobile workloads.

Of course the use of high-frequency mobile GPUs for higher performance results in dramatic increases in mobile power and energy consumption. For instance, Figure 2.1 shows the power consumption of a real mobile Adreno 225 GPU (shown in the upper dark gray) and the dual-core Snapdragon processors (shown in the lower light gray and the white for cores

Table 2.1: State-of-the-art Smartphone GPU

Devices	SoC	GPU	Max	Steps
MSM8960 MDP	Snapdragon S4	Adreno 225	400Mhz	4
Nexus 4	Snapdragon S4	Adreno 320	400Mhz	4
Galaxy S4	Exynos 5410	SGX544	480Mhz	4
Nexus 5	Snapdragon 800	Adreno 330	450Mhz	4
Galaxy S5	Exynos 542x	Mali-T628	578Mhz	5

0 and 1 respectively) for the MSM8960 Mobile Development Platform (MDP) executing the GLBenchmark Egypt graphics benchmark [5] on the High setting, to emulate an entirely GPU compute-bound test. Note the significantly higher mobile GPU power trace in dark gray (bouncing between 800mW and 1.2W) as compared to the much lower light gray and white traces representing the CPU core power consumption (bouncing between 100mW and 400mW). This figure clearly motivates the need to characterize mobile GPU workloads and develop more efficient mobile DVFS policies to save energy for graphics-intensive applications.

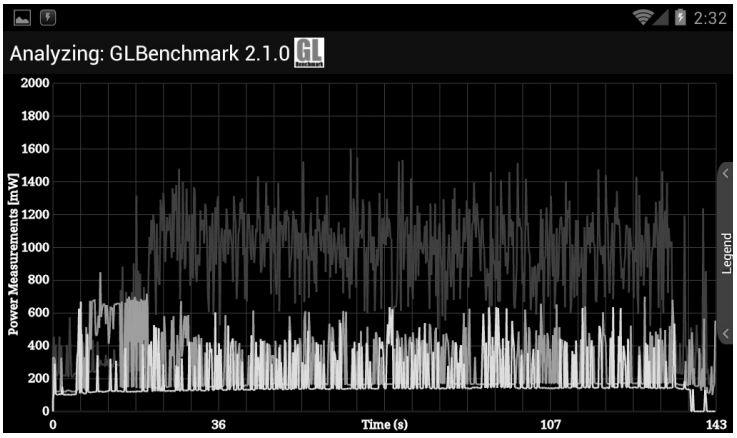


Figure 2.1: Mobile GPU Power Consumption [5]

Mobile platform vendors currently deploy proprietary mobile GPU DVFS techniques implemented in vendor-specific GPU device drivers, often hidden behind hardware security modules (e.g., the Nexus 4 GPU DVFS governor is hidden behind ARM's TrustZone). Note that existing mobile GPU DVFS policies are not workload-aware and furthermore do not allow for customization in the face of rapidly changing mobile GPU architectures. Further-

more, the overall energy efficiency of a mobile platform needs an integrated strategy that combines mobile GPU DVFS intelligently together with CPU DVFS for graphics rendering on mobile GPUs [22] [23], as well as memory bandwidth effects [30] [70] for high-performance graphics rendering applications with acceptance user experience. Such an approach needs a good understanding of mobile GPU graphics workload characterization for DVFS design, which to the best of out knowledge has not been addressed before.

In this chapter, we present a measurement study of mobile GPU graphics workload characterization for DVFS design to enable increased energy efficiency. We first introduce an abstracted mobile GPU pipeline and then correlate the OpenGL ES [53] pipeline on Android system to characterize performance and power metrics using custom micro-benchmarks that stress different components of the mobile GPU pipeline to model the effects of different application workloads on energy efficiency for graphics applications. We then propose opportunities for integrated mobile GPU-CPU DVFS design based on our observations and analyses.

2.1.1 Contributions

- Design and implementation of micro-benchmarks that stress specific graphics pipeline stages separately for graphics workload characterization.
- Observations and analysis of the micro-benchmark results which states the correlation between hardware characteristics and performance metrics
- Opportunities of average power and energy per frame for mobile DVFS design.

2.2 Motivation and Related Work

2.2.1 Motivation

Two major issues motivate our work: 1) the challenge of providing mobile GPU DVFS design concepts for better energy efficiency. 2) The need for thorough mobile GPU graphics workload characterization for mobile GPU DVFS design by analyzing the correlation between performance, power and energy efficiency for graphics-intensive applications.

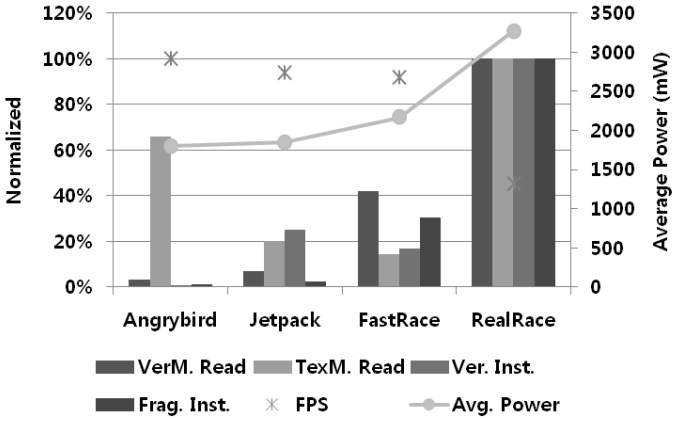


Figure 2.2: Motivating Example for GPU Workload Characterization.

Figure 2.2 outlines a motivating example for the mobile GPU pipeline where the X-axis shows a set of graphics-intensive games (Angrybird, Jetpack, FastRace, RealRace) profiled on the Nexus4 using the Adreno GPU profiler [85]. The left Y-axis shows the normalized Vertex/Texture memory read (MB/sec), the number of Vertex/Fragment shader instructions per second, and Frames-per-second (FPS). The right Y-axis shows the average power consumption for each game. This figure clearly shows that many different graphics workload factors affect the power consumption of these applications. For instance, Angrybird and Jetpack consume almost the same average power, but the profiled GPU workload characteristics are very different: while the most stressed workload factor for Angrybird is the texture memory read, Jetpack on the other hand is most stressed for executing vertex instructions. This example clearly shows that mobile GPU workload factors and workload variation sig-

nificantly affects the power consumption of different graphics pipeline components of mobile graphics-intensive applications, and opens up opportunities for better quality-aware DVFS policies that can exploit energy efficiencies of different GPU components. However as we describe below, current efforts have not undertaken a thorough study of mobile GPU graphics workload characterization to enable better quality-aware DVFS policies for energy efficiency.

2.2.2 Related Work

DVFS is an established energy conservation strategy that has been applied for both CPUs and GPUs in the mobile space as well as the desktop/server space. Before applying DVFS to graphics-intensive computer games, DVFS algorithms [3] [19] [47] [48] [62] [63] [102] for video decoding applications in the mobile space have largely been deployed because they are computationally expensive and its workload exhibits a high degree of variability [39].

With regard to DVFS for gaming workloads, gaming workload characterization for the performance and power consumption of desktop and mobile 3D games is prior to proposing DVFS policies. Some efforts characterized the performance and power consumption of desktop 3D games: static and dynamic workload characterization on graphics architecture features [21] [72] and analyses for a set of modern 3D games at the API call level and at the microarchitectural level [88], 3D graphics performance modeling [98], and the power consumption analysis and modeling of 3D graphics architecture [65] [91]. Ge et al. [30] introduced the impacts of DVFS on application performance and energy efficiency for GPGPU computing and compared them with DVFS for CPU computing; and Mei et al. [70] presented a measurement study that aims to explore how GPU DVFS affects the system energy consumption for GPGPU computation on desktop platforms. Our work is different from this large body of graphics (gaming) and GPGPU workload characterization in that here we analyze integrated mobile GPUs (which are architecturally different from mobile CPU graphics rendering and desktop GPUs [66]) by introducing customized micro-benchmarks

designed to stress individual mobile GPU components, and perform mobile graphics workload characterization in order to study the effects of different workload factors and workload variation. Using the results of these micro-benchmarks, we observe the correlation between performance, power, and energy efficiency to enable generation of enhanced mobile DVFS policies.

Some efforts have begun analyzing graphics-intensive rendering applications (e.g., 3D games) in mobile devices: Mallik et al. [68] presented power management for games by allowing the user to directly evaluate the current performance and scaling CPU frequency statically; Gu et al. [39][38] [37] [36] and Dietrich et al. [26] proposed CPU graphics rendering workload characterization and CPU DVFS for 3D Games, under the assumption that mobile devices such as PDAs and mobile phones do not have integrated mobile GPUs. Moreover, novel mobile GPU architecture design techniques [45] [14] for performance and energy-efficiency were proposed and [73] [64] characterized power consumption and performance of 3D mobile games, in addition to power analysis in a Smartphone [17]. With the emergence of high performance mobile GPUs, Dietrich et al. [22] [23] introduced CPU DVFS for mobile graphics rendering as an extension of [39] [38], but these work didn't focus on GPU DVFS, but rather on CPU DVFS for the mobile GPU. And, You et al. [100] explored to discover the potential of DVFS on embedded GPUs by analyzing workload variations of game application. Most recently, Pathania et al. [81] proposed an integrated CPU-GPU DVFS algorithm for power management for mobile games. However, they did not perform a detailed analysis and correlation between performance, power, and energy efficiency of the mobile GPU pipeline based on different workload factors and variations that could enable enhanced mobile DVFS design. In contrast, we present a thorough study of the correlations between mobile GPU hardware characteristics, workload factor variations, mobile GPU utilization, FPS, and consumed average power and energy per frame. In addition, we introduce opportunities for improved DVFS design of mobile GPU graphics rendering.

To the best of our knowledge, our work is the first to introduce quality-aware mobile

GPU graphics workload characterization considering better energy efficiency for DVFS design. With the recent emergence of high-performance mobile GPUs (with frequencies higher than 400Mhz and multiple frequency levels), we address the critical need for a comprehensive study of mobile graphics performance, power and energy efficiency using customized micro-benchmarks and varying graphics-intensive workload factors.

2.3 Graphics Workload Characterization

Unlike desktop GPUs, mobile GPUs are integrated in application processors with CPUs, and usually have a different rendering mode like Tile-Based Deferred Rendering (TBDR) [2] with narrow memory bandwidth and small numbers of processing elements. Since mobile GPUs are architecturally distinct from desktop GPUs, we begin by describing the mobile GPU hardware pipeline and illustrate graphic display with OpenGL ES on the Android system. We then present the design and implementation of our custom micro-benchmarks that are designed to stress different components of the mobile GPU pipeline.

2.3.1 Mobile Graphics Pipeline

2.3.1.1 Mobile GPU Hardware Pipeline

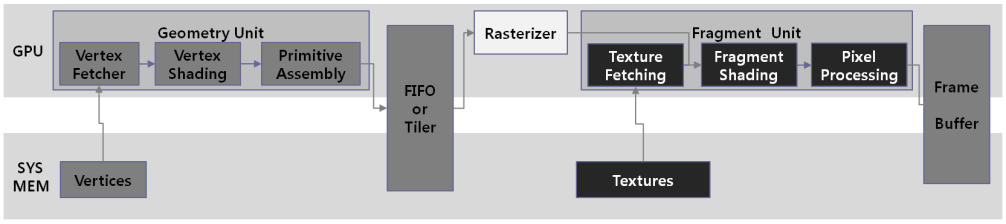


Figure 2.3: Abstract Mobile Graphics Pipeline.

Figure 2.3 shows a logical, abstracted GPU pipeline used by modern mobile GPUs (Mali, Adreno, and PowerVR GPUs) to explain the mobile graphics architecture and interaction

between GPU shaders and system memory. The three main components in the pipeline are the Geometry Unit, FIFO/Tiling Engine, and the Fragment Unit, as described below:

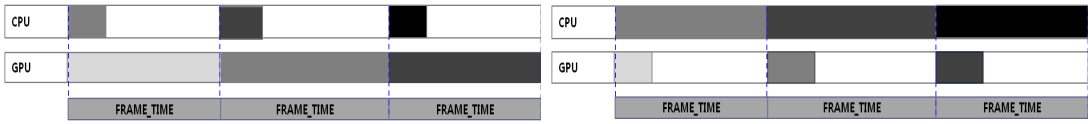
- The Geometry Unit is composed of three stages: i) the Vertex Fetcher reads the input vertices from memory, ii) the Vertex Shader transforms and shades the vertices, and iii) the Primitive Assembly stage assembles shaded vertices into triangles.

- The FIFO/Tiler Unit can be based on two rendering modes: immediate-mode rendering and tile-based rendering. In immediate-mode rendering, once a triangle has been transformed, it is immediately sent down to the graphics pipeline for further pixel processing. On the other hand for tile-based rendering, the Tiler Unit stores the triangles in memory and sorts them into tiles; since the transformed triangles from Geometry Unit have to be stored in memory and fetched back for rendering, there is a trade-off between memory traffic for geometry and memory traffic for pixels [15]. It should be noted that tile-based rendering is currently dominant in mobile GPUs.

- The Fragment Unit is mainly composed of programmable fragment shaders that process fragments generated by the rasterizer. Texturing usually happens here with the fetch of texture memory. The main function of the Fragment Unit is to process fragments, screen pixels and pixel processing such as reading and writing of color components and depth, and alpha blending.

2.3.1.2 OpenGL ES and Graphic Display

Figure 2.4 shows a high-level logical view of the integrated mobile CPU-GPU rendering pipeline model: the CPU produces graphics workload for mobile GPU to perform rendering operations. In this model, OpenGL ES applications have a temporal relationship with the execution of the underlying GPU hardware which performs rendering to fulfill application requirements.



(a) GPU Dominant Workload (b) CPU Dominant Workload

Figure 2.4: Logical Graphics Rendering Pipeline

At the highest (application) level, a series of OpenGL ES API calls is made by applications to indicate what to display according to the current application state. This procedure corresponds to the CPU stage of the pipeline in Figure 2.4. Then the rendering stages operate on the application graphics data to generate the display in the GPU state.

In Android, the SurfaceFlinger [9] window surface manager produces a frame by composing graphics data generated by different applications. Surfaceflinger then writes the outcome to framebuffer to display the composed image on the screen. This frame updating procedure corresponds to the dotted line in Figure 2.4. The frequency at which SurfaceFlinger updates displayed frames is labeled Frame Per Second (FPS), which is typically used as a metric to compare graphics performance (i.e., higher FPS is better graphics performance).

However, FPS is usually limited by the screen refresh rate since we cannot exploit an FPS larger than the screen refresh rate. In Android platforms, the process to update a frame is triggered periodically by the vertical synchronization (VSYNC) technique [34]. Generally the refresh rate is fixed at 60 Hz so that the maximum FPS is 60 and the frame time (reciprocal of FPS) is maximally limited to 16.67ms.

In the case of intensive graphics workload, the frame time increases to more than 16.67ms, so FPS drops below 60. With this observation, we are able to separate intensive mobile graphics rendering pipeline into two cases: a) GPU Dominant Workload, as illustrated in Figure 2.4(a) where the dominant frame time is the same with GPU execution time. In this case, the GPU is the bottleneck (e.g., the rendering is too complex to hit 60 FPS); and b) CPU Dominant Workload, as shown in Figure 2.4(b) where the dominant frame time is the same with CPU execution time. In this case, the bottleneck is not the GPU but the CPU (e.g., the corresponding CPU takes 30ms to produce a frame which the GPU takes 10ms to

render.)

2.3.2 Workload Characterization and Micro-benchmarks

We now present the design of our micro-benchmarks that can be used to analyze the correlation between performance, power and energy efficiency of the mobile GPU by stressing different components of the mobile graphics pipeline stages. Accordingly, we first categorize our micro-benchmarks from the perspectives of GPU memory-, GPU computation- and CPU computation-bound workloads.

2.3.2.1 Workload Characterization

GPU memory: Here we target GPU vertex memory fetch and texture memory fetch. Vertex memory contains vertex attributes such as positions, colors, etc. Vertices are the foundation of graphics objects hence they are necessary in all graphics rendering. On the other hand, texture memory is typically used in games/applications that require rich graphic details in objects and scenes. External memory access is very expensive to GPU performance, both due to longer memory latencies and also because it can decrease the available parallelism for GPU cores. Additionally, the GPU consumes significantly more power during the memory access. Thus, the GPU memory is an important component for characterization.

GPU computation: Vertex shaders and fragment shaders are considered in this category. Starting from OpenGL ES 2.0, programmable shaders are supported in mobile platforms. Compared to the fixed function pipeline in OpenGL ES 1.x, shader programs allow developers to control how the graphics objects are rendered. Since there may be a variety of shaders, we need to address their influence on performance and power.

CPU computation: Mobile applications can access hardware-accelerated graphics through

OpenGL ES commands. These commands are generated by algorithms run within applications to determine what to display in the next frame. As shown in Figure 2.4, graphics rendering includes the CPU as a whole. However, a complex programming model or complicated algorithms might reduce the FPS (resulting in quality loss) if excessive CPU time is spent on computation of OpenGL ES commands. Therefore, we also need to analyze the workload executed on CPU to understand its effect on graphics rendering.

2.3.2.2 Design and Implementation of Micro-benchmarks

TABLE 2.2 summarizes the Micro-benchmarks, their workload factor (i.e., which aspect of the mobile GPU pipeline is stressed), and the corresponding effects on the mobile GPU stages (i.e., Vertex Fetch, Vertex Shaders, Texture Fetch and Fragment Shaders). We briefly describe each Micro-benchmark below:

Table 2.2: Micro-benchmarks and their Pipelined Workloads

MBs	Workload Factor	CPU	Vertex Fetch	Vertex Shaders	Texture Fetch	Fragment Shaders
mb-VerM	Vertex Memory Read	Only GL API	# of Vertices	Minimized	None	None
mb-TexM	Texture Memory Read	Only GL API	Fixed	Minimized	Texture Img Size	Minimized
mb-VerSh	Vertex Instructions	Only GL API	Fixed	Ver. Shader P.G	None	None
mb-FragSh	Fragment Instructions	Only GL API	Fixed	Minimized	None	Frag. Shader P.G
mb-App	CPU Exec. Time	GL API + Busy Loop	Fixed	Minimized	None	Minimized

mb-VerM: stresses the vertex fetcher (start of the GPU pipeline) by giving different number of vertices to vary the amount of vertex data read from main memory. The workload of following pipeline stages might be affected due to different amounts of vertex data. Therefore, in order to minimize additional workload in the following stages, a simple vertex shader program is used into the later pipeline stages where the vertices will be discarded by the clipping test.

mb-TexM: targets the amount of texture memory read from main memory. Both vertex and fragment shaders should support texture mapping in order to do texture fetch. The amount of texture memory read is varied by applying different sizes of texture images. Since we aim to evaluate only the amount of texture memory fetch, we use only fixed vertex data

with fixed texture coordinates (instead of diverse texture mapping techniques).

mb-VerSh: targets vertex shaders that perform coordinate transformation of vertex data. *mb-VerSh* stresses vertex shaders by changing the number of vertex instructions with a vertex shader program. The program computes the lighting effect on vertex color and repeats itself redundantly with a different number of iterations to increase the workload of vertex shader. *mb-VerSh* also adopts the same strategy as *mb-VerM* to disable the following pipeline stages.

mb-FragSh: stresses the Fragment shaders that manipulate data in each fragment such as color and texture. As with Micro-benchmark *mb-VerSh*, we apply a fragment shader program that supports lighting effect on fragment color, and increase the number of fragment instructions by repeated execution of this program to stress the fragment shader.

mb-App: simulates the higher application workload and the OpenGL ES API calls at the application level. The workload is stressed by modifying the time spent by CPU with a configurable code stub. In order to overcome varying execution times (due to effects of the CPU frequency governor and load balancer in Linux), we apply a fixed CPU frequency to emulate a consistent execution time for the same *mb-App* configuration.

2.4 Experimental Setup and Results

2.4.1 Experimental Setup and Methodology

TABLE 2.3 summarizes our platform configuration where we use a Nexus 4 device installed with Android version 4.2.2 and Linux build 3.4.0. As shown in Figure 2.5, we use the Adreno Profiler [85] to get the profiled data of workload factors such as texture memory read per second and vertex instructions per second, GPU utilization, and FPS. The Adreno Profiler is

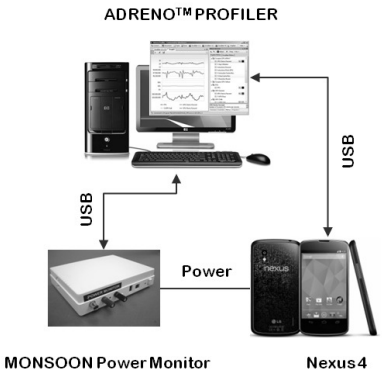


Figure 2.5: Power Measurement and Adreno Profiler Setup

a PC based application with mobile graphics optimization and debugging features to identify performance bottlenecks in each application, and users can dynamically make modifications to see how they affect performance at runtime. The device runs with 40% of screen bright-

Table 2.3: Platform Configuration

Feature	Description
Device	Nexus4 [57]
SoC	Snapdragon S4 Pro APQ8064 [6]
CPU	Cortex-A15 MP (Quad-core), 1.5Ghz
GPU	Adreno 320, 400Mhz
System RAM	2GB RAM (533Mhz Dual-ch.)
Mem. Bandwidth	up to 8.5GB/sec
OS(Platform)	Android 4.2.2
Linux Kernel	3.4.0
Profiler	Adreno Profiler Version 3.6 [85]

ness and airplane mode where the power consumption is measured using a Monsoon power monitor [74]

On the Nexus 4, the Adreno GPU is used with a maximum frequency of 400Mhz, and supporting four frequency steps (128Mhz, 200Mhz, 325Mhz, and 400Mhz). The Nexus 4 smartphone runs a power and thermal management process in the background that dynamically controls the CPU and GPU frequency for thermal throttling when a threshold temperature is reached. To avoid interference with our experiments, we disabled this process to obtain consistent experimental data.

We measured and profiled this system using the Micro-benchmarks. For each micro-benchmark, we repeat the experiments by using different combinations of workload factors and GPU frequencies. The power measurement is conducted separately from profiling since the connection between the PC and the phone would lead to imprecise results. Workload factors are adjusted by the configurations within the benchmark and then installed as a new application to run on the phone. The GPU frequency is changed through the *sysfs* interface [1] provided by GPU device driver.

2.4.2 Experimental Results

We now present the results and analysis of running the five different micro-benchmarks for workload characterization.

2.4.2.1 Micro-benchmark mb-TexM

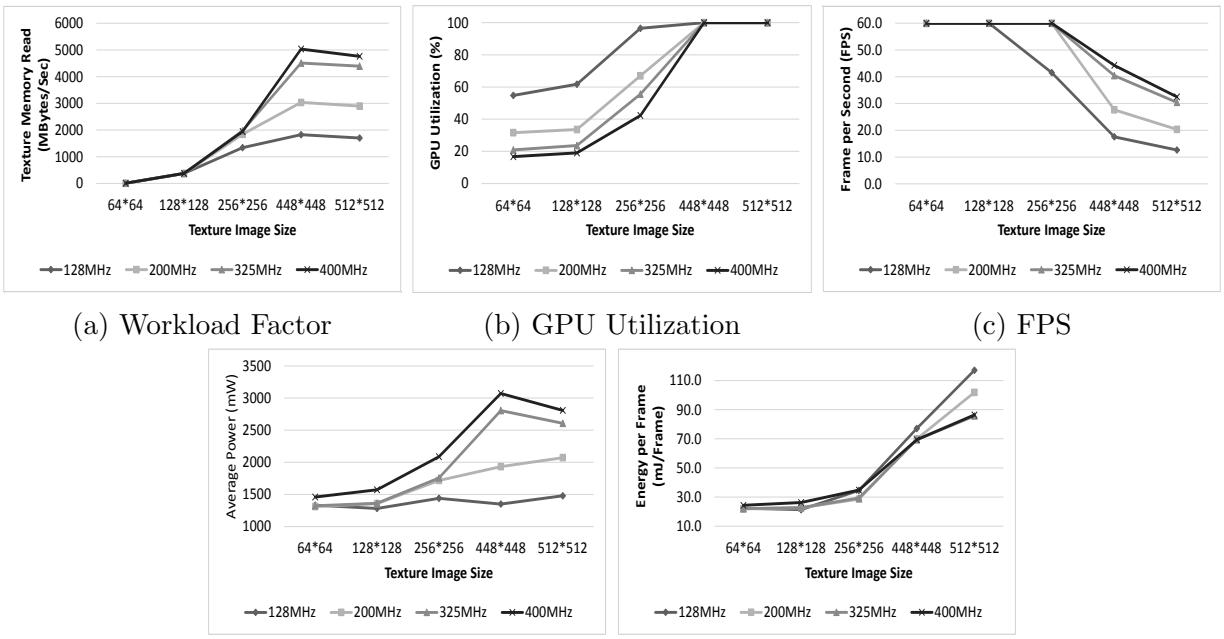
Figure 2.6 shows the results of *mb-TexM* at different frequencies and workload variation. Based on these results, we analyze and observe the following patterns and trends. As shown in TABLE 2.4, in order to make workload variation, the amount of texture memory read is varied by applying five different texture image sizes.

Table 2.4: Workload Variation in *mb-TexM*

Tex. Image Size	64*64	128*128	256*256	448*448	512*512
Tex. Mem. R.(MB/s)	8	376	1,945	5,032	4,763

Table 2.5: Percentage of Texture L2 Cache Miss in *mb-TexM*

% Tex. L2 Miss	64 * 64	128 * 128	256 * 256	448*448	512*512
128MHz	0.39	7.39	15.78	23.15	25.18
200MHz	0.39	7.30	14.97	23.50	25.40
325MHz	0.39	7.37	16.00	23.86	25.67
400MHz	0.39	7.36	15.81	24.16	26.01



(a) Workload Factor (b) GPU Utilization (c) FPS

(d) Average Power (e) Energy per Frame

Figure 2.6: Results of mb-TexM at Different Frequencies with Workload Variation

Workload Factor: As shown in Figure 2.6(a), the amount of texture memory read is directly proportional to the increased size of texture images except the maximum image size. In this case, the amount of texture memory read is decreased in spite of the increased size. According to the profiling results, the peak texture memory read is 5.032GB/s in the 448 image size but the peak texture L2 cache miss is highest in the maximum image size as shown in TABLE 2.5. (Ideally the peak memory bandwidth on Nexus4 is 8.5GB/s, but the highest memory throughput was 5.032GB/s among the results of our micro-benchmarks.). Based on these results, we speculate the reason that the maximum image size has worse memory throughput than the 448 image size. The memory bandwidth is already exhausted(saturated) in the 448 image size and image size is enlarged additionally, resulting in the increase of the texture L2 cache miss and subsequent increase in the memory latency. Therefore, memory throughput is decreased in the maximum image size because memory throughput is proportional to frequency but is inversely proportional to the memory latency.

The amount of texture memory read at different frequencies are the same up to the 256 image while they are proportional to the frequency in the 448 image size and the maximum

image size. The main reason is that FPS has substantially reduced by using low GPU frequencies in spite of 100% GPU utilization for all frequencies. In other words, the number of frames rendered by GPU are reduced accordingly so that the amount texture memory read are also decreased.

GPU Utilization: Figure 2.6(b) shows that the GPU utilization is proportional to the increased size of texture images except the maximum image size. In contrast with the Workload Factor(texture memory read), GPU Utilization is inversely proportional to frequency below 128 image size and remains the same above 448 image size.

FPS: As shown in Figure 2.6(c), FPS is inversely proportional to the increased size of texture images. From a frequency, FPS is the same below 128 image size and directly proportional to GPU frequency (except 400Mhz) above 448 image size. The peak memory bandwidth limitation in 400Mhz results in more FPS reduction in texture memory fetching compared to other frequencies.

Average Power: As shown in Figure 2.6(d), the average power is proportional to the increased size of texture images except for the maximum image size. Below 128 image size, there is a considerable gap of average power consumption between 325Mhz and 400Mhz compared with other frequencies. For instance, the average power is reduced from 1573mW to 1361mW in the 128 image, (i.e., 13.45% of the average power of 400Mhz). Because it's already in the maximum FPS, energy efficiency could be improved up to 13.45% without performance reduction by GPU DVFS. On the other hand, a trade-off between power and performance exists above 448 image size.

Energy per Frame[1]: As shown in Figure 2.6(e), in the range below 128 image size, 400Mhz has the highest energy per frame (the worst energy efficiency). However, energy per frame in the range above 448 image size is very diverse according to different workload. In

[1]Since the execution time remains the same for FPS at a fixed frequency, we assume in the rest of this chapter that the energy per frame is proportional to the power consumed.

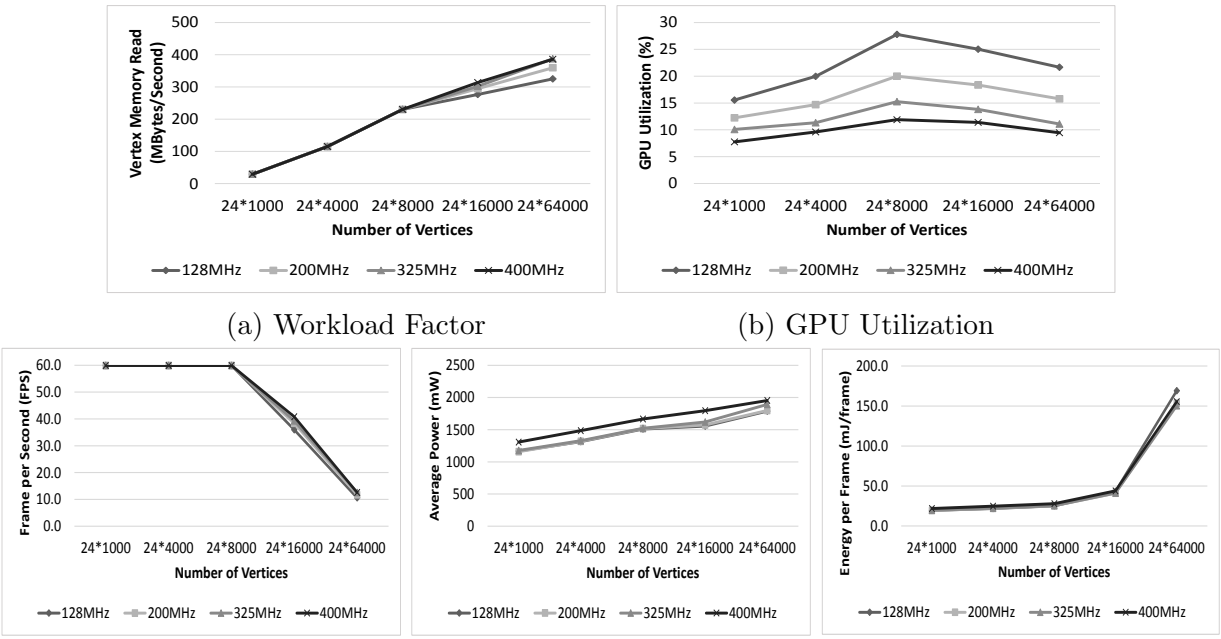
the case of the 448 image size, only energy per frame of the lowest frequency is worse than other frequencies because the power consumption in 128Mhz is relatively higher compared to the frequency ratio while FPS is directly proportional to the frequency ratio (i.e., execution time is inversely proportional to the frequency ratio.). However for the maximum image size, the trend shows that higher frequency has lower energy per frame (better energy efficiency).

In particular, the difference of energy per frame among lower frequencies in 512*512 is higher than that of higher frequencies. This is related to the different rate of increase of FPS and average power according to each frequency because we measured the total system power instead of the GPU power consumption. (i.e., the rate of increment of FPS is faster than that of average total power). We will explain the reasons in detail in Chapter 2.4.2.4.

2.4.2.2 Micro-benchmark mb-VerM

Workload Factor: As shown in Figure 2.7(a), the amount of vertex memory read is changed by the number of rendered vertices in *mb-VerM*. TABLE 2.6 shows an example of how it varies. *mb-VerM* aims to emulate one type of GPU memory-intensive applications. Intuitively, the bottleneck of performance would not be the system memory bandwidth. The results show that the performance is not bounded by the bandwidth as the maximum vertex memory throughput is 386MB/s, which is far less than system theoretic memory bandwidth 8.5GB/s. Therefore, we speculate that too many vertices (i.e., vertex data per frame) results in vertex processing bottleneck with the additional needs of internal buffers.

GPU Utilization: Unlike previous results, the utilization shown in Figure 2.7(b) implies that the amount of vertex memory read does not influence utilization directly. As previously mentioned, GPU still performs operations in vertex shader. Therefore, GPU computation workload slightly changes as number of vertices increases. In cases that the number of vertices are below or equal to 24*8000, the utilization increases along with the number of vertices. In other cases, we observe that the percentage of clock cycles where the GPU cannot make



(a) Workload Factor (b) GPU Utilization

(c) FPS (d) Average Power (e) Energy per Frame

Figure 2.7: Results of mb-VerM at Different Frequencies with Workload Variation

any more requests for vertex data increases while fetching vertex data, leading to a drop in utilization.

Table 2.6: Workload Variation in mb-VerM

Num. of Vertices	24*1000	24*4000	24*8000	24*16000	24*64000
Ver. Mem.(MB/s)	29	115	230	314	386

FPS: From Figure 2.7(c), we see that the number of vertex memory read has not saturated but FPS starts to drop when the number of vertices are increased to 24*16000. We speculate that since the geometry stage and fragment stage in GPU pipeline execute sequentially, the time spent in geometry stage becomes the bottleneck for vertex memory intensive cases, which leads to low FPS. Though vertex memory read throughput is affected by GPU DVFS as shown in Figure 2.7(a), FPS does not benefit significantly from high GPU frequency.

Average Power: Unlike the previous results that power consumption reduces largely along with decreased FPS, it remains a near-linear relationship with the amount of vertex memory read. The effects of GPU DVFS in vertex memory read throughput is insignificant,

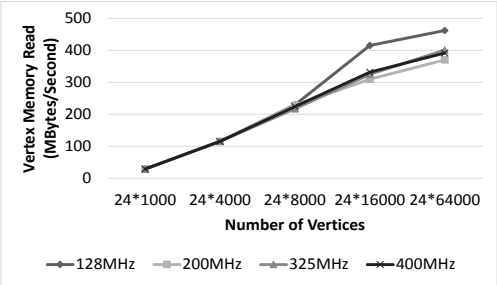


Figure 2.8: Workload Factor result with Enabled CPU DVFS on mb-VerM

which results in similar FPS and power usage pattern in different frequencies.

Energy per Frame: Energy per frame shown in Figure 2.7(e) with each frequency in different workload factors are very similar, which indicates again that GPU DVFS does not improve energy efficiency evidently in vertex memory-intensive applications.

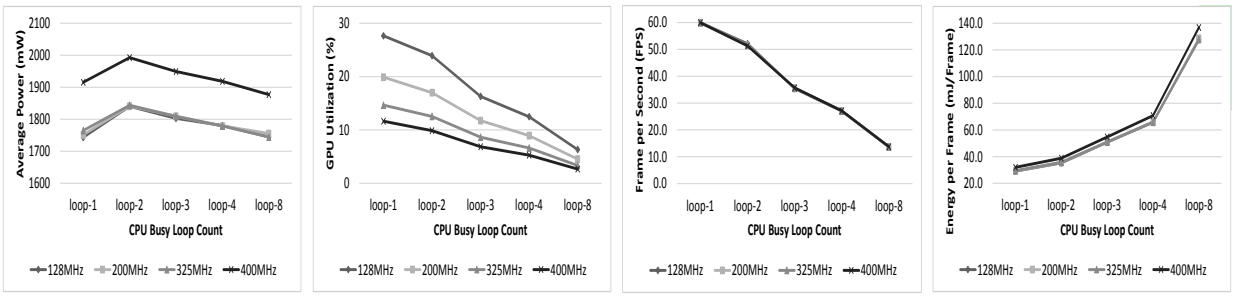
CPU DVFS effect on *mb-VerM*: We also observe that CPU DVFS results in unstable vertex memory throughput. Compared with Figure 2.7(a) (which runs with fixed CPU frequency), Figure 2.8 shows the results running with the default ondemand CPU frequency governor. The ondemand governor scales CPU frequency up and down depending on CPU utilization periodically which results in the unstable vertex memory throughput. In other words, vertex memory throughput also largely depends on CPU frequency.

2.4.2.3 Micro-benchmark mb-App

Workload Factor: The execution time spent in CPU is controlled by a configurable code stub. GPU rendering operations for the next frame are executed after CPU has done the simulated workload. TABLE 2.7 shows the CPU execution time variation in different configurations.

Table 2.7: Workload Variation in mb-App

Workload	loop-1	loop-2	loop-3	loop-4	loop-8
Exec. Time	9.39ms	19.11ms	29.31ms	37.22ms	75.17ms



(a) Average Power (b) GPU Utilization (c) FPS (d) Energy

Figure 2.9: Results of mb-App at Different Frequencies with Workload Variation

GPU Utilization: Figure 2.9(b) shows the GPU utilization of *mb-App*. Though GPU utilization of different frequencies have the same pattern with previous results, the utilization goes low as workload increases. The reason is because when CPU spends more time in application logic, GPU has more time to be idle which causes low GPU utilization.

FPS: In Figure 2.9(c), the result of *loop-1* shows both CPU and GPU could complete the workload in time to maintain highest FPS as the refresh rate. In other cases, frame time is dominated by CPU execution time, hence FPS reduces as CPU workload increases. Note that while FPS is low, GPU utilization is also low, indicating that performing GPU DVFS with only GPU utilization metric might lead to unexpected results in FPS. Therefore, the CPU utilization should also be considered.

Average Power: From the case of *loop-1* to *loop-2* in Figure 2.9(a), CPU becomes fully busy, therefore the average power increases. In other cases, CPU consumes the same power as it is already fully utilized; and the average power drops along with FPS reduction because fewer frames are rendered by GPU. Similar to previous micro-benchmarks results, the power consumption in 400MHz is significantly more than those for other frequencies. However, GPU DVFS does not affect FPS at all. A general observation is that if CPU execution time dominates the frame time, it is beneficial to scale down GPU frequency to the lowest allowable level such that GPU execution time does not dominate the frame time. Therefore, GPU DVFS governor should include CPU utilization to make decisions for low

power consumption.

Energy per Frame: As GPU DVFS has no effect on FPS, the energy per frame increases with frequency, which leads to the observation that GPU DVFS governor should scale down frequency to the lowest possible level.

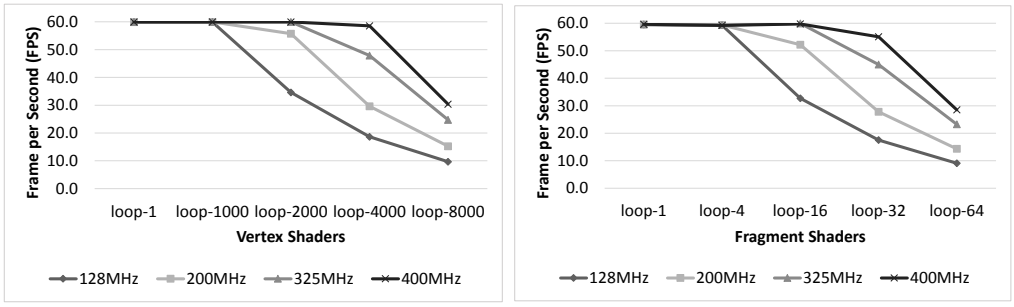
2.4.2.4 Result and Analysis on mbVerSh and mbFragSh

TABLE 2.8 shows the workload variation in the two micro-benchmarks respectively. *mb-VerSh* stresses vertex shaders by changing the number of vertex instructions using a vertex shader program and *mb-FragSh* uses a similar approach to stress fragment shader.

Table 2.8: Workload Variation of *mb-VerSh* and *mb-FragSh* Analysis

Num. of loop	loop-1	loop-1000	loop-2000	loop-4000	loop-8000
Ver. Inst.(M/s)	1.3	674	1,348	2,632	2,731
Num. of loop	loop-1	loop-4	loop-8	loop-16	loop-32
Frag. Inst.(M/s)	1,555	2,890	4,713	8,349	14,377

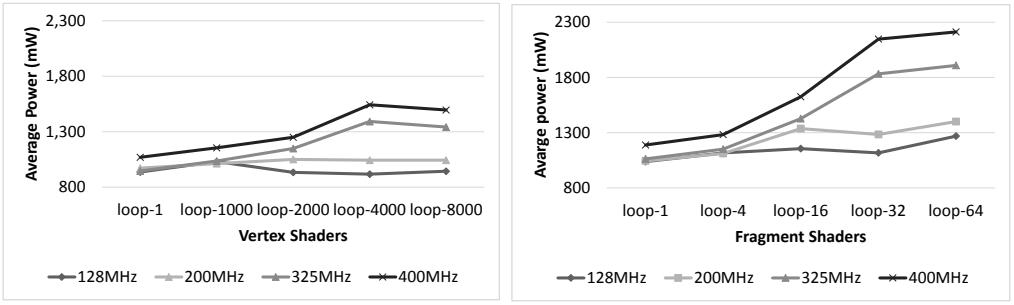
FPS: As shown in Figure 2.10(a) and Figure 2.10(b), patterns of the results in both *mb-VerSh* and *mb-FragSh* are very similar to *mb-TexM*. We observe that FPS is inversely proportional to the increased workload but is proportional to frequency below the maximum FPS for all frequencies. Compared with *mb-TexM*, the main difference here is that the performance is not limited by memory bandwidth.



(a) FPS (b) FPS

Figure 2.10: FPS of *mb-VerSh* and *mb-FragSh*

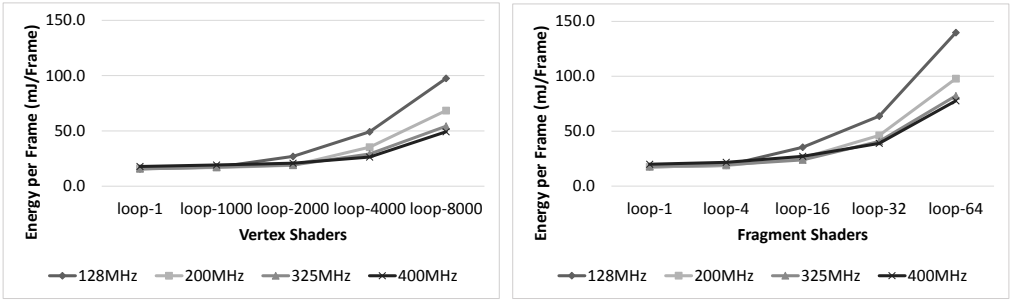
Average Power: The power consumption in *mb-VerSh* and *mb-FragSh* have similar pattern but the values are different while they perform at the same FPS. For instance, the average power of 400Mhz of the maximum workload in Figure 2.11, where the FPS is around 30, the power consumption is 1496mW in *mb-VerSh*, and 2147mW in *mb-FragSh*. Referring to Figure 2.6(d), the power consumption is 2807mW in *mb-TexM* at the same situation. Therefore, *mb-TexM* is most influential in power consumption among the three micro-benchmarks due to excessive external memory access.



(a) Average Power (b) Average Power

Figure 2.11: Average Power of *mb-VerSh* and *mb-FragSh*

Energy per Frame: As shown in Figure 2.12(a) and Figure 2.12(b), the patterns of energy per frame in *mb-VerSh* and *mb-FragSh* are also similar to *mb-TexM*.



(a) Energy per Frame (b) Energy per Frame

Figure 2.12: Energy per Frame of *mb-VerSh* and *mb-FragSh*

In mb-TexM, mb-VerSh and mb-FragSh, energy per frame has a similar pattern that in high workloads where FPS is below 60 in all frequencies, we observe the difference of energy per frame becomes obvious in different GPU frequencies as the workload factor increases.

We first explain how the difference comes from. As we measured total system power instead of GPU power, it doesn't increase as much as the FPS increases while scaling up the GPU frequency. It leads to the results that energy per frame is different in each frequency and higher frequencies have lower energy per frame than lower frequencies.

In addition, we now explain why the difference of energy per frame becomes larger while increasing workload factors. First, the power consumption remains the same after the GPU is saturated by increasing workload factors ideally. For example, Fig. 11 shows that the power consumption in the two right-most configurations are very similar. Hence, we can assume that the ratio of power consumption in different frequencies is the same under different workload factors. Second, as we mentioned before, the ratio of FPS is almost the same with the ratio of GPU frequencies. Suppose we want to compare the energy per frame of two frequencies f1 and f2. We can assume that the FPS of f1 is x and the FPS of f2 is c*x where c is a constant. The total system power consumption of f1 is y and that of f2 is k*y where k is a constant. By the energy per frame equation: Energy per frame = Average Power / FPS, we could find out that the difference of energy per frame in the two frequencies is proportional to y/x. Because y remains the same in high workloads, it turns out that the difference is proportional to the reciprocal of x, which is the FPS. As the workload factor increases, the FPS drops in all frequencies. Therefore, the difference becomes large.

2.5 Opportunities for DVFS design

Our experimental results and analysis of micro-benchmarks gives us insights to identify several opportunities for improved DVFS design for GPU graphics rendering. It becomes clear that an integrated DVFS strategy addressing the entire graphics pipeline – including GPU DVFS and CPU DVFS – is necessary to achieve energy efficiency without loss of quality. In particular, among our five micro-benchmarks, we observe that *mb-TexM*, *mb-VerSh* and

mb-FragSh are dependent on GPU DVFS. Both *mb-VerM* and *mb-App* are less dependent on GPU DVFS compared with the other micro-benchmarks. For *mb-VerM*, we observe that vertex memory read bandwidth is influenced by CPU DVFS.

2.5.1 GPU DVFS

- Hints for GPU frequency settings to save power

Our experimental observations can exploit opportunities for energy-efficient GPU DVFS when there are variations in power consumption at different frequencies, based on GPU hardware characteristics and the available GPU frequency configuration steps. For example, the Adreno 320 GPU using the four frequency steps on Nexus4 has the potential for power saving mainly between the frequency settings of 325Mhz and 400Mhz.

- Different patterns between FPS and average power

With regard to performance, FPS is directly proportional to the ratio of GPU frequency if there is no bottleneck (e.g., memory bandwidth limitation). However, the power patterns at different frequencies are not similar to FPS because the average power consumption is more dependent on GPU hardware characteristics and workload variation of applications compared to FPS. From our experiments, we observe that power prediction is more difficult and irregular than performance prediction for workload variations. Thus a heuristic approach using semantic information for power management in DVFS design may be more influential for accurate power prediction.

- Energy efficiency patterns for different workload variations

From the perspective of energy efficiency (i.e., energy per frame) on the Nexus 4, we observe that the highest frequency is worse than all the other frequencies for low workload.

On the other hand, high frequencies have low energy per frame (better energy efficiency) for high workloads which will have below the maximum FPS for all frequencies. Therefore, the lowest allowable frequency (without performance degradation) will be the optimal choice for low workloads, but the highest allowable frequency (with power increase but lower energy per frame and better performance) would be the optimal choice for intensive workload.

2.5.2 CPU DVFS

- CPU dominant workload (CPU execution time is the frame time)

In the case of CPU intensive workloads, GPU DVFS does not affect the FPS, average power, and energy per frame at all. For instance, *mb-App*'s frame time is the same as CPU computation time, thus GPU frequency should be at the lowest allowable setting. In this situation, the CPU is the bottleneck and the GPU is idle with the result that graphics rendering performance and average power is directly affected by CPU DVFS, not GPU DVFS.

- GPU dominant workload (GPU execution time is the frame time)

Conversely, when the frame time is dominated by GPU execution time, CPU DVFS should be utilized to improve energy efficiency. For example, the CPU has lots of idle time while handling the intensive workload cases in *mb-VerSh* or *mb-FragSh*. Therefore, a CPU DVFS which is aware of the graphics rendering pipeline should be able to improve energy efficiency in GPU-bound cases. In this situation, the GPU is the bottleneck and the CPU is idle. In order to improve energy efficiency, we have opportunities of CPU maximum frequency capping or CPU hot-plug techniques besides CPU DVFS.

2.5.3 Integrated DVFS

- CPU-GPU integrated DVFS

As mentioned earlier, the entire graphics pipeline spans the range from CPU invocation of OpenGL ES API calls to GPU execution of the graphics rendering pipeline. While traditional approaches apply CPU and GPU DVFS separately, we believe that the energy efficiency can be further improved by integrated CPU and GPU DVFS from the perspective of workload characteristics. In both GPU dominant workload and CPU dominant workload, CPU-GPU DVFS or CPU DVFS considering graphics renderings will be beneficial for better energy efficiency.

- Integrated DVFS considering memory bandwidth

Another important observation is that memory bandwidth depends on the micro-architecture of CPU which is similar to the results in [90]. The bandwidth varies while performing CPU DVFS which might lead to unexpected graphics performance in terms of FPS. Therefore, for intensive graphics applications like 3D games, these applications exhibit various combinations of CPU-, GPU-, and memory-bound workloads. Thus an integrated CPU and GPU DVFS approach should also consider the memory bandwidth effect concurrently to handle a wider spectrum of applications.

2.6 Summary

In this chapter, we conducted an experimental study of GPU graphics workload characterization on a commercial mobile phone (Nexus 4) to study the effects of DVFS from the perspective of performance, power and energy efficiency for executing mobile graphics-intensive applications. We outlined a methodology for creating micro-benchmarks that stress specific

stages within the abstracted graphics pipeline, and demonstrated experimental results by executing these micro benchmarks and profiling various metrics with the goal of opening up new opportunities for integrated GPU, CPU and memory DVFS for energy efficiency without loss of quality.

Our experimental results on micro-benchmarks uncovered several observations. From the perspective of energy per frame (mJ/Frame), the highest frequency is worse than other frequencies for low workloads; in other words, the lowest frequency that does not incur performance reduction will be the optimal choice. On the other hand for intensive workloads, higher frequencies result in lower energy per frame (better energy efficiency) in addition to improved performance. Another observation is that for better energy efficiency using DVFS, the effects of CPU, GPU and memory on GPU graphics rendering should be considered as a whole. Memory impact should be specifically addressed because memory DVFS has not been adopted in mobile systems as well as CPU DVFS might also alter memory bandwidth according to micro-architectures. Therefore, we believe that there is potential for even better energy efficiency, especially when the entire mobile graphics pipeline (comprising the CPU, GPU and memory) is considered holistically.

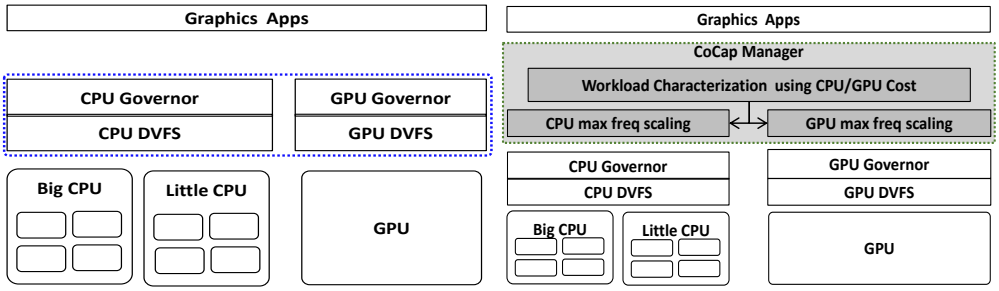
Chapter 3

Cooperative CPU-GPU Frequency Capping

3.1 Introduction

Mobile platforms are increasingly demanding high performance and longer battery life at the same time resulting in the move towards Heterogeneous MultiProcessor Systems-on-Chip (HMPSoC) for high performance, coupled with various software governors to achieve energy efficiency through DVFS. For instance, the Exynos 5 (5422) [89] HMPSoC integrates ARM's big.LITTLE [35] Octa multi-core CPU and ARM's Mali-T628 MP6 GPU on the same chip. To achieve energy efficiency, traditionally independent CPU and GPU DVFS power management techniques are common for commercial platforms, and some recent efforts have proposed integrated CPU-GPU governors [81] [80] [51] for energy efficiency, but have focused on a small set of mobile games that have specific workload characteristics. (Figure 4.1(a)).

Traditional DVFS approaches suffer from two drawbacks: 1) they are not able to achieve energy efficiency across a wide range of mobile games that exhibit varying CPU and GPU



(a) General Mobile System (b) Proposed Co-Cap System

Figure 3.1: System Comparison

workloads, and 2) these approaches are customized for specific mobile platforms. To address both of these issues, in this chapter we present Co-Cap (Figure 4.1.(b)), a methodology that achieves energy efficient DVFS across a wide range of mobile games, and that is also easily portable to newer mobile architectural platforms on top of the default CPU and GPU governors.

Device	Release	SoC	CPU	Max. F. (Ghz)	GPU	Max. F. (Mhz)
Nexus 4	Feb. 12	Snapdragon S4	Quad	1.5	Adreno 320	400
Galaxy S4	Mar. 13	Exynos 541x	Octa b.L	1.2/1.6	SGX 544	480
Nexus 5	Oct. 13	Snapdragon 800	Quad	2.3	Adreno 330	450
Galaxy S5	Mar. 14	Exynos 542x	Octa b.L	1.3/1.9	Mali-T628	543
Nexus 6	Oct. 14	Snapdragon 805	Quad	2.7	Adreno 420	600
Galaxy S6	Mar. 15	Exynos 7420	Octa b.L	1.5/2.1	Mali-T760MP8	772

Figure 3.2: Mobile Platform Trends.

Mobile platforms face the dual challenges of rapidly changing (heterogeneous) architectures, and the continuing emergence of a plethora of mobile games. For instance, Figure 3.2 shows the progression of the Nexus and Samsung Galaxy series platform architecture, demonstrating the rapid changes in (heterogeneous) processor, and GPU configurations. New energy efficient DVFS governors have to be developed rapidly for each new architectural family. On the other hand, at the application level, we are seeing a range of mobile games that exhibit diverse CPU and GPU workloads, that require widely different strategies for achieving energy efficiency while delivering acceptable game performance. For instance, Figure 3.3 shows that each game application can be located in a specific quadrant of a CPU/GPU workload intensity matrix. Angry Birds has very low CPU and GPU workloads, while GFX

benchmark is GPU-bound and Jetski Race is very CPU-bound; some applications such as GPUbench are more balanced in terms of CPU and GPU workloads. For these four different types of graphics workloads, we define them as No CPU-GPU dominant, CPU dominant, GPU dominant, CPU-GPU dominant workloads respectively.

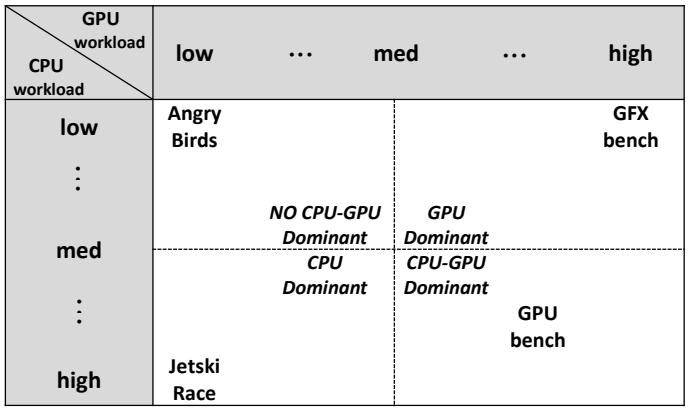


Figure 3.3: Different types of CPU/GPU Workload.

The traditional approach of separate DVFS governors for CPU and GPU are unable to achieve good energy efficiency while delivering acceptable performance across this wide range of mobile game applications. Even recent efforts [82] [80] [51] using integrated CPU-GPU governors are only targeted towards specific classes of mobile games. Furthermore, these approaches do not provide a methodology for easily porting their strategies across newer architectural platforms that appear in rapid succession (e.g., Nexus and Galaxy series in Figure 3.2).

To address these dual issues, this chapter presents Co-Cap, an energy-efficient cooperative CPU-GPU frequency capping strategy for mobile games. This work makes the following specific contributions:

- We present a methodology for cooperative CPU-GPU frequency capping to achieve energy efficiency for a diverse set of mobile benchmarks and games
- Our capping strategy avoids unnecessarily higher frequency of CPU and GPU considering both FPS and per-frame energy saving on top of the default CPU and GPU governors

- We present characterization of diverse mobile graphics gaming workloads to enable efficient dynamic application of frequency capping
- We introduce a systematic methodology with completeness using different types of benchmark sets and fine-grained refinement steps for configuration of frequency capping lookup tables
- We demonstrate efficacy of Co-Cap across over 200 micro-benchmarks and 40 real mobile graphics applications, using a representative training and deployment set achieving significant improvement (up to 27.6%) energy efficiency with minimal loss in performance

3.2 Related Work

DVFS is a traditional energy conservation strategy that has been applied for both CPUs and GPUs, but traditionally in the desktop space [102] [62]. [65] proposed a statistical approach for power consumption analysis and modeling on desktop GPU and [64] characterized performance and power consumption of 3D mobile games. However, their work did not consider DVFS design but rather focused on power consumption and performance issues. [30] introduced the impacts of DVFS on application performance and energy efficiency for GPGPU computing and compared them with DVFS for CPU computing; and [70] presented a measurement study that aims to explore how GPU DVFS affects the system energy consumption for GPGPU computation on desktop platforms. And, [55] proposed a GPGPU power model and integrated the power model with the cycle-level simulator GPGPU-Sim and demonstrated the energy savings by utilizing dynamic voltage and frequency scaling (DVFS) and clock gating. Our work is different from this large body of desktop GPU/GPGPU workload characterization in that here we focus on integrated mobile GPUs (which are architecturally different from desktop GPUs [66]). For graphics-intensive applications such as mobile games,

typically frames per second (FPS) and energy per frame (or FPS per watt) are used as performance and energy consumption metrics, respectively.

Some efforts have begun analyzing graphics-intensive rendering applications (e.g., 3D games) in mobile devices: [39] and [38] proposed CPU graphics rendering workload characterization and CPU DVFS for 3D Games, under the assumption that mobile devices such as PDAs and mobile phones do not have integrated mobile GPUs; and With the emergence of high performance mobile GPUs, [22] [23] [25] [24] introduced CPU DVFS for mobile graphics rendering as an extension of [39], [38]; however these efforts didn't focus on GPU or cooperative CPU-GPU DVFS schemes, but addressed CPU DVFS for mobile GPU graphics rendering. Moreover, You et al. [100] and [101] explored to discover the potential of DVFS on embedded GPUs by analyzing workload variations of game application and proposed a QoS-aware GPU DVFS policy that ensures a consistent GPU performance under the QoS constraint; and Kim et al. [54] proposed a GPU power model for smartphones using graphics rendering applications.

In our previous work [75], we developed micro-benchmarks for mobile graphics workload characterization, analyzed the results of benchmarks, and introduced several opportunities for improved DVFS design of mobile GPU graphics rendering, but did not present specific DVFS strategies. [94] and [83] proposed the power management for CPU-GPU heterogeneous SoC architectures, but they focused on the general purpose computing applications, not gaming applications. [82] proposed an integrated CPU-GPU DVFS algorithm for power management for mobile games. However, their work didn't consider quantitative evaluation for energy saving (e.g., per-frame energy or FPS per watt); [80] also proposed a power-efficient integrated CPU-GPU DVFS strategy by developing power-performance models predicting the impact of DVFS on mobile gaming workloads; [51] presented a queuing model to represent the interaction between CPU, GPU, and Display; [18] proposed a behavior-aware integrated CPU-GPU power management approach using the system-call and OpenGL API information for mobile games; [20] proposed an adaptive on-line CPU-GPU governor for

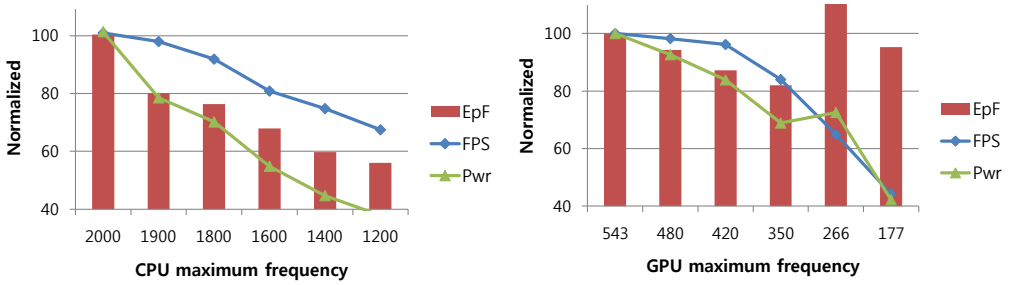
games on mobile devices to minimize energy consumption. However, their work was applied to a specific set of games exhibiting a narrow range of CPU-GPU workloads; and they did not show applicability across a wide range of games exhibiting diverse CPU-GPU workloads (Figure 3.3) in spite of significantly different results from different types of graphics workloads. [67] presented a perception-aware adaptive scheme that sets the resolution during game play for power savings, not DVFS techniques; and [84] proposed a control-theory based dynamic thermal management technique that cooperatively scales CPU and GPU frequencies to meet the thermal constraint while achieving high performance for mobile gaming, but they mainly focused on a dynamic thermal management (DTM) algorithm rather than coordinated CPU-GPU DVFS power management.

In this work, we complement shortcomings of previous efforts [75] by proposing Co-Cap. Our work is fundamentally different from previous integrated CPU-GPU DVFS techniques in that our approach dynamically scales the maximum frequency of CPU and GPU using frequency capping lookup tables according to the normalized CPU and GPU cost on top of the default CPU- and GPU governors. Although frequency capping for energy efficiency was initially introduced in [58], their approach was restricted to the CPU governor; in contrast to the best of our knowledge, our work is the first to introduce a coordinated CPU and GPU maximum frequency capping technique that achieves energy efficiency (lower energy per frame) across a diverse range of mobile games while delivering acceptable performance (FPS).

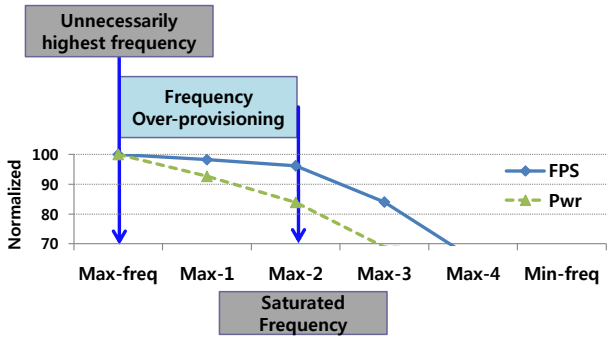
3.3 Motivation

Mobile platforms pose a challenge for simultaneous reduction of energy consumption while delivering acceptable performance across a wide range of mobile gaming applications. As motivating examples, Figure 3.4.(a) and 3.4.(b) show the normalized FPS, power consump-

tion (Pwr), and energy per frame (EpF) on one CPU dominant (Q3Zombie Map4) and the other GPU dominant (Action Bike) game benchmarks by changing CPU and GPU maximum frequencies allowing dynamic frequency changes under standard governors implemented in Linux. Even though FPS declines little until 1900Mhz in Figure 3.4(a) and until 420Mhz in Figure 3.4(b), the power consumption is dramatically reduced compared to the FPS reduction. This shows that limiting the maximum frequencies that can be reached during dynamic frequency scaling gives significant opportunity for performance and power consumption optimizations in modern CPU/GPUs. In other words, we can exploit this frequency over-provisioning to achieve energy saving up to 20% in Figure 3.4.(a) and up to 15% in Figure 3.4.(b) within little FPS (3%) decline comparing with the default governors by scaling CPU and GPU maximum frequencies. We call the frequency beyond which the FPS degrades as "saturated frequency" (Figure 3.4(c)). By identifying these saturated frequencies after characterization of CPU- and GPU graphics workloads, we can save energy with minimal performance degradation by eliminating CPU and GPU frequency over-provisioning.



(a) Q3Zombie Map4 (b) Action Bike



(c) The Saturated Frequency

Figure 3.4: Motivating Examples

Moreover, we adopt a CPU and GPU frequency capping heuristic for the following reasons: 1) Simplicity: adding a capping module on top of the default (or custom) CPU/GPU governors is easier than complicated integrated CPU/GPU governors. 2) Portability: capping is easily adaptable to newer platforms. Compared to the proposed integrated CPU-GPU governors [80] [51], our CPU-GPU cooperative frequency capping methodology is easily portable in terms of prospective architectural hardware limitations and governor software complexity. In our work, we tested two real platforms (from Nexus4 [57] to Odroid-XU3 [43]), which have totally different chip vendors, CPU and GPU software governors. First, if a governor is hidden behind in a secure hardware like ARMs TrustZone (e.g., GPU governor on Nexus4), their approaches may have serious limitations/challenges in implementation even if there is no problem in the model building. Second, we scale only CPU and GPU maximum frequencies using simple lookup tables on top of the default CPU and GPU governors. If we consider a total platform change (e.g., Nexus4 to Odroid-XU3), our methodology is easily portable in terms of design and implementation time of an integrated governor, for rapidly changing mobile platforms. 3) Elimination of wasteful frequency over-provisioning: commercial CPU governors typically scale frequency using utilization based thresholds. In other words, if CPU utilization is greater than or equal to a certain threshold, the governor sets the highest frequency from the corresponding frequency [58]. This is also similar in GPU governor but more conservative. However, in both cases these approaches may result in frequency over-provisioning, since the frequency is set to a higher level even when the target QoS has been met at maximum utilization. This results in wasted power.

3.4 Co-Cap Methodology

Co-Cap orchestrates cooperative CPU-GPU dynamic maximum frequency capping (scaling) by avoiding frequency over-provisioning of CPU or GPU considering both performance (FPS)

and per-frame energy saving on top of the default CPU- and GPU governors. Co-Cap has two phases: a training phase and a deployment phase as shown in Figure 3.5.

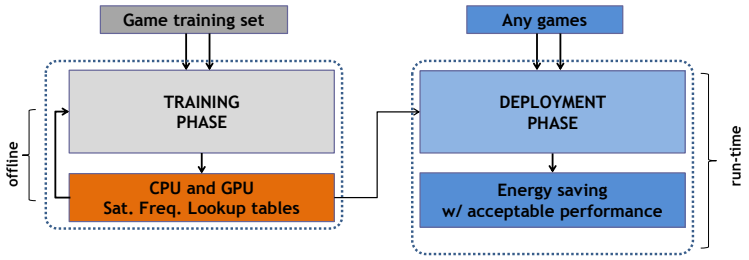


Figure 3.5: Co-Cap Overview.

In the training phase, we build CPU and GPU saturated frequency lookup tables offline using training sets. Then in the deployment phase, Co-Cap uses these saturated frequency lookup tables at runtime to set the appropriate CPU and GPU frequency caps based on the characteristics of the executing mobile benchmark.

3.4.1 Training Phase

As shown in Figure 3.6, the training phase is composed of three steps: data capturing step, saturated frequency lookup table building step, and fine-grained refinement step. The main objective of the training phase is to build the saturated frequency lookup tables where the maximum frequency values for both CPU and GPU can be obtained by CPU and GPU workload cost indices.

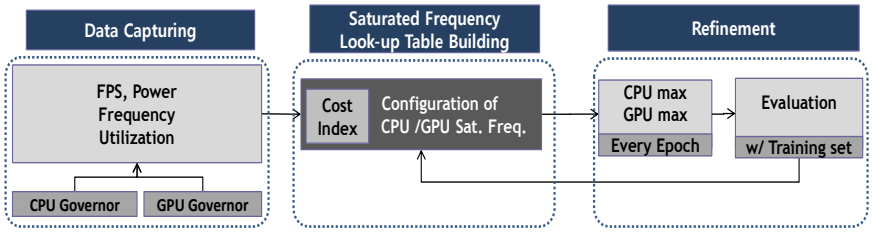


Figure 3.6: Co-Cap Training Phase.

3.4.1.1 Data Capturing Step

As shown in Table 3.1, all data including frequency and utilization can be captured dynamically from each software component of CPU and GPU governors, and power sensor driver

(or using power monitor).

Table 3.1: Captured Data

Category	SW component	Metrics Data
CPU	CPU Governor	per-cpu Utilization, Frequency
GPU	GPU Governor	Utilization, Frequency
Power	Power Sensor Driver	CPU, GPU, DRAM
Perf.	Android and GPU Governor	FPS

The sample training set shown in Figure 3.7 includes micro-benchmarks that cover different quadrants of the CPU-GPU workloads shown in Figure 3.3. These workload variations are typically quantified using a cost metric that is a product of the utilization and frequency [16]. Accordingly, we deploy normalized CPU- and GPU costs as shown in Equation(1), where the cost is defined as the product of the processor current average utilization and its current average frequency divided by the product of the maximum utilization and its maximum frequency to have the normalized range of 0 to 100. For CPU utilization, the highest CPU utilization among CPU cores is used according to the assumption that there is usually one graphics rendering thread mainly affecting the graphics performance for most mobile graphics applications, and the utilization of the thread is mostly highest among threads.

$$Normalized\ Cost = \frac{Curr\_Util.\ \times\ Curr\_Freq.}{Max\_Util.\ \times\ Max\_Freq.} \tag{3.1}$$

And then, CPU/GPU saturated frequency lookup tables (LUTs) are configured gradually through the following Building Step and Fine-grained Refinement Step.

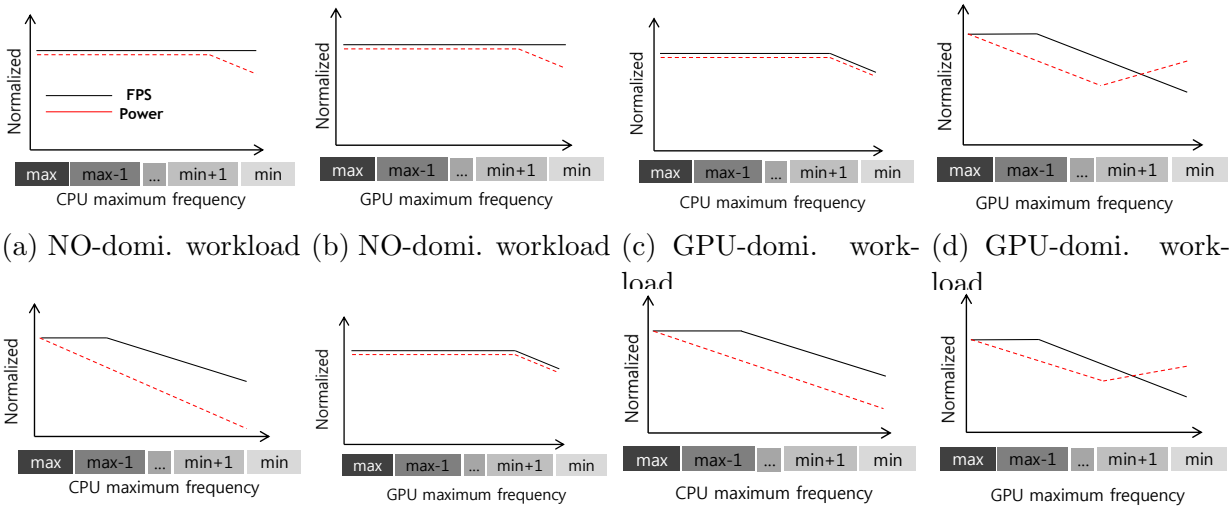
3.4.1.2 Frequency Lookup Table Building Step

Using the normalized CPU and GPU costs, we determine the number of CPU/GPU cost index (N) of a training set, which is ultimately used as the cost index of each entry (0 to N-1) in the saturated frequency lookup tables. First, a training set covers the four basic categories by CPU- or/and GPU-dominance (i.e., No, CPU, GPU and CPU-GPU domi-

nant workloads); and then, for example, each category has more detailed workload segments corresponding to low, medium and high workloads as shown in Figure 3.7, the number of CPU/GPU cost index (N) will be 6 (0 to 5) and the number of the total entries will be 36.

CPU\GPU Cost Index	0 (Low)	1 (Med)	2 (High)	3 (Low)	4 (Med)	5 (High)
0 (Low)	NO-mb1	NO-mb2	NO-mb3	GPU-mb1	GPU-mb2	GPU-mb3
1 (Med)	NO-mb4	NO-mb5	NO-mb6	GPU-mb4	GPU-mb5	GPU-mb6
2 (High)	NO-mb7	NO-mb8	NO-mb9	GPU-mb7	GPU-mb8	GPU-mb9
3 (Low)	CPU-mb1	CPU-mb2	CPU-mb3	CGU-mb1	CGU-mb2	CGU-mb3
4 (Med)	CPU-mb4	CPU-mb5	CPU-mb6	CGU-mb4	CGU-mb5	CGU-mb6
5 (High)	CPU-mb7	CPU-mb8	CPU-mb9	CGU-mb7	CGU-mb8	CGU-mb9

Figure 3.7: A Sample of the Training Set.



(a) NO-domi. workload (b) NO-domi. workload (c) GPU-domi. workload (d) GPU-domi. workload

(e) CPU-domi. workload (f) CPU-domi. workload (g) CGU-domi. workload (h) CGU-domi. workload

Figure 3.8: Effects of CPU (or GPU) maximum frequency capping on FPS and Power.

Now, let's assume that we use a new game on a new platform. By observing the CPU and GPU utilization and frequency for a specific amount of time, the game will be located in a specific entry having the corresponding CPU and GPU cost index, which has energy-efficient CPU and GPU saturated frequencies.

Before describing how to configure the saturated frequency look-up tables, we need to explain the general trend of maximum frequency set-up for the four different application

quadrant categories shown in Figure 3.3: No CPU-GPU (No) dominant, CPU dominant, GPU dominant, and CPU-GPU (CGU) dominant. Figure 3.8.(a) and (b) show the performance/power consumption graph trends of No-dominant workloads. FPS remains the same (i.e., the 60 maximum FPS) for all CPU and GPU frequencies, which means that the saturated frequencies of CPU and GPU are available up to the minimum frequency of CPU and GPU for energy saving without FPS reduction. Figure 3.8.(c) and (d) show the performance/power consumption pattern of GPU dominant workloads. For GPU frequency, FPS is flat until max-1 frequency and power consumption reduces gradually until min+1 frequency. Therefore, in order to achieve energy saving with little FPS degradation, the GPU saturated frequency can be reduced to max-1 frequency. For CPU capping frequencies, FPS and power consumption are almost similar except the minimum capping frequency, which means that the CPU saturated frequency should be higher than the minimum frequency in order to prevent FPS reduction. Figure 3.8.(e) and (f) show the performance/power consumption of CPU dominant workloads. Here FPS is almost similar until max-1 frequency and power consumption gradually decreases with CPU capping frequency drop, which means that the CPU saturated frequency could be available up to the max-1 frequency. For GPU capping frequencies, FPS and power consumption are almost similar except the minimum capping frequency, which means that the GPU saturated frequency should be higher than the minimum frequency in order to prevent FPS reduction. Finally in Figure 3.8.(g) and (h), CGU-dominant workload is the combination of CPU dominant workload and GPU dominant workload. Therefore, in CPU and GPU, max-1 frequency can be configured to the saturated frequency.

Using this characterization process, the saturated frequency of each application in the

training set can be configured appropriately as shown in Equation (2):

$$
\begin{aligned}
&Saturated\ Frequency = max(F_{fps}, F_{lp}) \\
&where\ F_{fps} = lowest\ maximum\ frequency \\
&with\ minimal\ (< 3\%) FPS degradation \\
&F_{lp} = frequency\ for\ lowest\ power\ consumption
\end{aligned}
\tag{3.2}
$$

In other words, from the captured data of each application, we choose the higher frequency among the lowest maximum frequency having little FPS (up to 3%) decline and the lowest maximum frequency having lowest power consumption.

LUTs after the Building Step: In this stage the LUTs are generated automatically with the help of our automatic measurement tool (AMT) through the LUT Building Step with the ability to capture data automatically.

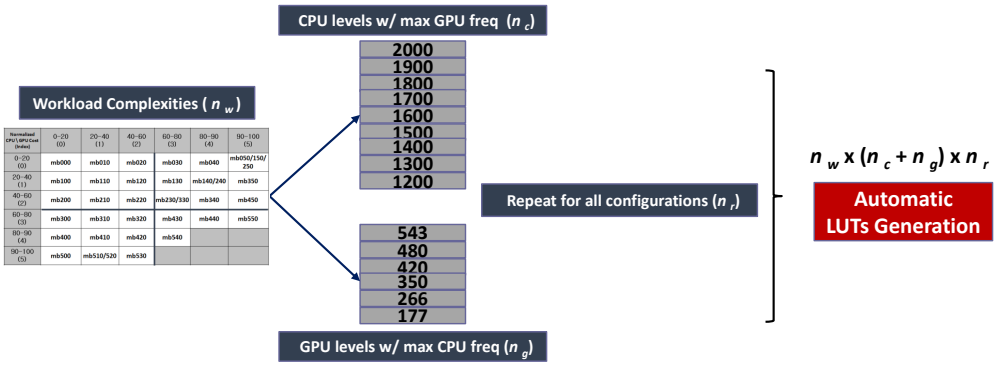


Figure 3.9: The Proposed Methodology for Data Collection and LUT Building.

As shown in Figure 3.9, we first use a basic 36-benchmark training set for various workload complexities covering all LUT entries (at least one benchmark per entry). Then, we sweep the CPU frequency across the set of frequencies supported by the target system (9 frequencies in CPU) at the maximum GPU frequency of 543 Mhz and the GPU frequency (6 frequencies in GPU) at the maximum CPU frequency of 2.0 Ghz. (In our experiments,

we collect data for 36 x (9 + 6) = 540 different configurations at least 3 times.) Finally, using the data of each corresponding benchmark (complexity), the CPU/GPU saturated frequencies in each entry are configured by using the Equation (2) with a FPS-target (in this study a FPS-target of 97% is used for minimal FPS degradation among user-definable FPS-targets).

CPU \ GPU Cost Index	0	1	2	3	4	5
0	1200	1200	1200	1200	1200	1200
1	1200	1200	1200	1200	1200	1200
2	1200	1200	1200	1200	1200	1200
3	1200	1200	1200	1200	1200	1500
4	1700	1700	1700	1600	-	-
5	1900	1900	1900	-	-	-

(a) CPU LUT

CPU \ GPU Cost Index	0	1	2	3	4	5
0	177	266	350	420	480	480
1	177	266	350	420	480	480
2	177	266	350	420	480	480
3	177	266	350	420	480	480
4	177	266	350	420	-	-
5	177	266	350	-	-	-

(b) GPU LUT

Figure 3.10: LUTs after the Building Step

Figure 3.10 shows the results of the LUTs after the building step. However, these preliminary LUTs should be refined because it is hard to use directly offline-built LUTs for all possible (diverse and dynamic) applications without evaluation. Therefore, we refine the LUTs through the fine-grained refined step using evaluation results of diverse types of benchmarks.

3.4.1.3 Fine-grained Refinement Step

The lookup tables generated statically in the building step need to be refined using the results of runtime evaluations because it is hard to use directly the offline-built lookup tables for all possible (diverse and dynamic) applications without evaluation. Moreover, the refinement step is composed of two steps as shown in Figure 3.11: 1) from using low-variation workload benchmarks. 2) to using high-variation workload benchmarks.

Through these two steps, we use a greedy heuristic to configure the best of lookup tables. This is the assumption that the global best configuration can be reached by a series of locally best choices (i.e., in the refinement of lookup tables, the locally best saturated frequency

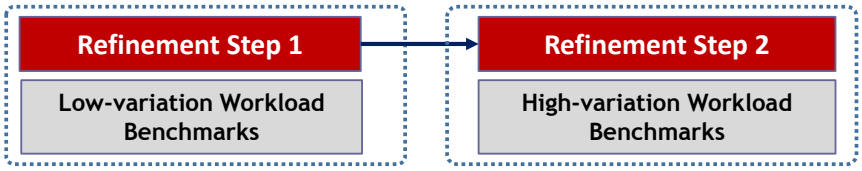


Figure 3.11: Fine-grained Refinement Step.

configuration at each entry with the hope of finding the global best of lookup tables). For the cases when there are two or more saturated frequencies in a entry, the maximum saturated frequency is chosen using the upper-bound approach.

Methodology of Fine-grained Refinement Step: Our fine-grained refinement step strategy is outlined in Figure 3.12. In Step1, we start the refinement using the 36-benchmark

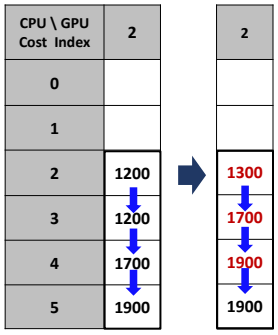
```
FPS_t is FPS-target in the refinement step
S is a Training Set of benchmarks
Each benchmark i has a FPS_i (average FPS during the evaluation time),
CI_Def_i (average cost index in the default) and
CI_CC_i (average cost index in Co-Cap) after using the LUTs
```

```
Step 1.  FOR each benchmark i in S DO
         if FPS_i is less than FPS_t
         Add benchmark i to A

Step 2.  While A is not yet empty DO
              select a benchmark i with highest FPS drop   ←  {greedy choice}

Step 3.       FOR every related cost index entry, from CI_CC_i to CI_Def_i DO
   Step 3.1.     Increase the frequency step by step
   Step 3.2.     IF the cost entry moves THEN set the last frequency before the entry move
   Step 3.3.     ELIF FPS increase gradually THEN set the last frequency and stop
   Step 3.4.     ELSE stop
```

Figure 3.12: The Methodology of Fine-grained Refinement Step.

set (a set of low-variation benchmarks) that covers all cost index entries; if the average FPS of each benchmark (FPS_i) after using the LUTs in the building step is less than a predefined FPS-target (FPS_t), we set apart the benchmarks as a new set A based on the assumption that if the FPS-drop of the benchmark is within the FPS-target, the current saturated frequency in the entry is appropriate for the corresponding benchmark. In Step2, among the new set A we select a benchmark that has the highest FPS-drop using a greedy heuristic for the reduction of the total refinement time. Note that each benchmark of the new set A should have this step iteratively according to the descending order of FPS-drop using the repeatedly refined LUTs; if the highest FPS-drop entry is refined as early as possible with appropriate saturated frequencies, sequentially the refinement time of the remaining benchmarks will be

reduced. In Step3, we refine the related cost index entries (from CI_CC_i to CI_Def_i) using the each selected benchmark and the order of CPU or GPU LUT refinement is determined by the cost dominance of the related entries. From the Co-Cap cost entry (CI_CC_i), we increase the frequency step by step (Step 3.1). As the frequency goes up, if there is a cost entry change, the last frequency in the previous entry is set to a new saturated frequency of that entry (Step 3.2). In the case of no entry change, there are two more conditions: if there are gradual FPS increases due to the frequency increases, we set the last frequency as a new saturated frequency and stop the refinement of the benchmark (Step 3.3); otherwise (i.e, no FPS increases) we stop the refinement of the benchmark without additional saturated frequency updates (Step 3.4). And gradually, we refine the LUTs after changing a benchmark set from low-variation to high-variation benchmarks using the same procedure (from Step 1 to Step3).



(a) Example (mb530)

	Refinement	Cost Index	C / GPU Cost	FPS (vs. Def)
Def	-	(5,2)	96,57	44.18 (100)
CC	-	(2,2)	57,40	31.94 (72.3)
1	**1200** -> **1300**	2,2 → 3,2	60,40	32.34 (72.3)
2	**1200** -> 1300	3,2	65,41	32.81 (73.2)
3	1300 -> 1400	3,2	66,41	33.21 (74.3)
4	1400 -> 1500	3,2	71,49	36.66 (75.2)
5	1500 -> 1600	3,2	76,51	38.29 (83.0)
6	1600 -> **1700**	3,2 → 4,2	80,49	39.25 (86.7)
7	**1700** -> 1800	4,2	86,49	38.9 (88.0)
8	1800 -> **1900**	4,2 → 5,2	90,53	42.7 (96.7)
9	1900 (max in LUT)	5,2		

(b) Detailed Refinement for the Related Cost Index Entries

Figure 3.13: Detailed Fine-grained Refinement Step.

To illustrate further, we describe the third step with a specific example in Figure 3.13. The benchmark is mb530 which has CI_Def_i of (5,2) and CI_CC_i of (2.2) with 72.3% FPS of the default; the related entries of this benchmark are (2,2), (3,2), (4,2) and (5,2) which are mostly CPU-dominant (i.e., CPU cost index is bigger than GPU cost index) entries. Therefore, we first start the CPU LUT refinement from the (2,2) entry. As the frequency goes up (from 1200 to 1300), the last frequency (1300) is set to a new saturated frequency in (2,2) entry because the cost entry changes from (2,2) to (3,2). And then the last frequency

1700 (from 1200 to 1700) for the changed (3,2) entry and the last frequency 1900 (from 1700 to 1900) for the changed (4,2) entry are set successively. (For the mb530 benchmark, Steps 3.3 and 3.4 were not exercised, but some benchmarks such as mb450 and mb550 do use Steps 3.3 or 3.4).

In addition, if there is continuous maximum utilization during a certain amount of time, the saturated frequency should be scaled up dynamically in order to prevent FPS reduction. We deploy a heuristic where, if there is successive (e.g., for 2 or 3 epochs) extreme CPU or GPU utilization (e.g., 100%), the saturated frequency is dynamically scaled up to the one-level higher CPU and/or the two-level higher GPU frequency. In order to determine the thresholding values, we used empirical observations. For the number of successive epochs, we compare the results of FPS and EpF of one of training sets. Based on the observations, the interval of 2 epochs is better than that of 3 epochs in terms of FPS and EpF and two-level higher GPU frequency is more appropriate than one-level higher frequency in terms of preventing sudden FPS reduction.

LUTs after Fine-grained Refinement Step: Figure 3.14 shows the final saturated frequency LUTs configured as output of the training phase. For the No-dominant workload, available lowest frequencies were configured. For the CPU-dominant workload, CPU saturated frequency is almost near to the maximum CPU frequency (i.e., 1700 - 1900Mhz), and GPU saturated frequency is similar to or higher than the minimum frequency (i.e., 177 - 325Mhz). For the GPU-dominant workload, GPU saturated frequency is almost near to the maximum frequency (i.e., 420 - 480Mhz), and CPU saturated frequency is similar to or higher than the minimum frequency (i.e., 1200 - 1300Mhz). Finally, for the CGU-dominant workload, lower frequencies than those of CPU-dominant workloads were configured.

Evaluation of Fine-grained Refinement Step: The main objective of this step is to refine the lookup tables generated statically in the building step (Figure 3.10) using benchmark sets with evaluations. This is because it is hard to use directly the offline-

CPU \ GPU Cost Index	0	1	2	3	4	5
0	1200	1200	1200	1200	1200	1200
1	1200	1200	1200	1200	1200	1300
2	1300	1300	1300	1300	1300	1300
3	1700	1700	1700	1500 *1600	1500 *1600	1500 *1600
4	1900	1800 *1900	1900	1600	-	-
5	1900	1900	1900	-	-	-

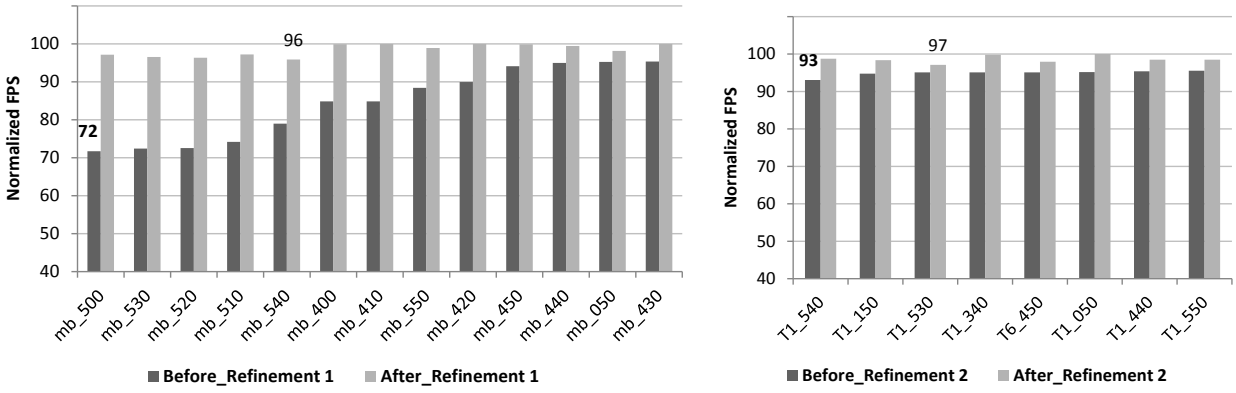
(a) CPU LUT

CPU \ GPU Cost Index	0	1	2	3	4	5
0	177	266	350	420	480	480
1	177	266	350	420	480	480
2	177	266	350	420	480	480
3	177	266	350	420	480	480
4	177	266	350	420	-	-
5	177	266	350	-	-	-

(b) GPU LUT

Figure 3.14: Final LUTs after the Refinement Steps (*frequencies are refined during the refinement step 2)

built lookup tables for all possible applications. We use two evaluation sets–one set of low-variation (composed of mixed workloads of adjacent 3-4 entries) and the other set of high-variation (composed of mixed workloads of several entries) workload benchmarks–using the same methodology described in Figure 3.12. (More detailed explanations for low- and high-variation benchmarks will be described on the benchmark sets in Chapter 3.5.1).



(a) Evaluation of the Refinement Step 1

(b) Evaluation of the Refinement Step 2

Figure 3.15: The Evaluation of the Fine-grained Refinement Steps

First, Figure 3.15 shows the evaluation results after the refinement step 1 using the set of low-variation benchmarks. When we use the offline-built LUTs (Figure 3.10), the results show that 13 benchmarks out of 36 benchmarks do not achieve a FPS target (97% FPS of the baseline) having 84% FPS on average and that the highest FPS drop is 28% in mb_500 benchmark. With the LUTs after the refinement step 1 (Figure 3.14), the results show that

34 benchmarks achieve the FPS target having 98% FPS on average and that the highest FPS drop is only 4% in mb_540 benchmark. Second, when we evaluate the LUTs after the refinement 1 using another set of high-variation workload benchmarks, the results show that 8 benchmarks out of 60 benchmarks still do not achieve a FPS target (97% FPS of the baseline) having 95% FPS on average and that the highest FPS drop is 8%. Using the LUTs after the refinement step 2 (*frequencies in Figure 3.14), the results show that all benchmarks achieve the FPS target having 98.6% FPS on average.

In summary, to configure the best of LUTs, we used a greedy heuristic based on the assumption that the global best configuration can be reached by a series of locally best choices at each entry. Using our systematic methodology of the fine-grained refinement step with two different types of benchmark sets, we evaluated that our final LUTs can achieve a target FPS with more reduced maximum frequency (i.e., lower power consumption) compared to the baseline for high-variation as well as low-variation workload benchmarks.

3.4.2 Deployment Phase

In the deployment phase (Figure 3.16), data capturing, maximum frequency setting, and evaluation steps can be executed at runtime using the CPU and GPU saturated frequency lookup tables. This ensures evaluation of applications across a wide range of games exhibiting diverse CPU and GPU workloads.

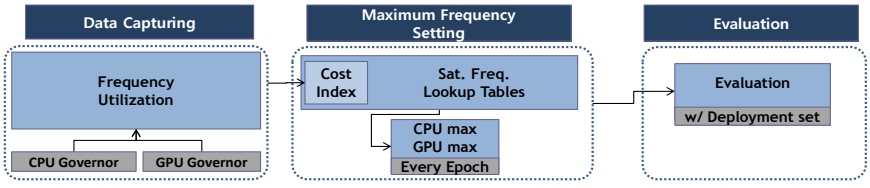


Figure 3.16: Co-Cap Deployment Phase.

Data Capturing Step: This step is exactly the same as the training phase (all data except power consumption as shown in Table 3.1 can be captured dynamically in every

epoch).

Maximum Frequency Setting Step: In this step, the corresponding CPU and GPU maximum frequencies for next epoch dynamically chosen from the CPU and GPU saturated frequency lookup tables (configured offline in the training phase) after calculation of CPU and GPU cost index at the end of the current epoch using the results of data capturing step.

Evaluation Step: We evaluate applications of the deployment set, using the detailed evaluation methods as described in the experimental setup of the next chapter.

3.5 Evaluation of Co-Cap

3.5.1 Experimental Setup

We evaluate Co-Cap on the ODROID-XU3 development board installed with Android 4.4.2 and Linux 3.10.9; Table 3.2 summarizes our platform configurations.

Table 3.2: Platform Configuration

Feature	Description
Device	ODROID-XU3 [43]
SoC	Samsung Exynos5422 [89]
CPU	Cortex-A15 2.0Ghz and Cortex-A7 Octa-core CPUs
GPU	Mali-T628 MP6, 543Mhz
System RAM	2Gbyte LPDDR3 RAM at 933MHz
Mem. Bandwidth	up to 14.9GB/s
OS(Platform)	Android 4.4.2
Linux Kernel	3.10.9

The ODROID platform is equipped with four TI INA231 power sensors [93] measuring the power consumption of high-performance big CPU cluster (CPU-bc), energy-saving little CPU cluster (CPU-lc), high-performance MP6 GPU and memory respectively. The CPU supports cluster-based DVFS at nine frequency levels (from 1.2Ghz to 2.0Ghz) in CPU-bc and at seven frequency levels (from 1.0Ghz to 1.6Ghz) in CPU-lc, and GPU supports six

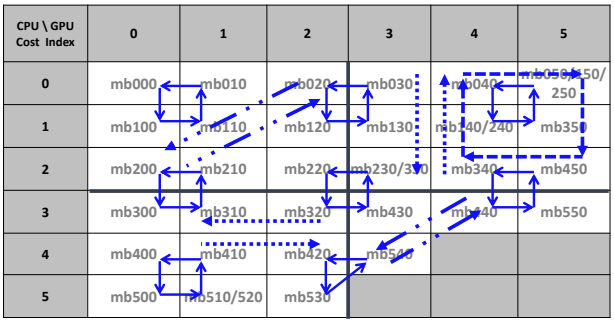
frequency levels (from 177Mhz to 543Mhz); the default governors are interactive CPU governor [13] and ARM's Mali Midgard GPU governor [60].

Benchmark Sets: We deployed an extensive set of benchmarks for our experiments composed of over 200 micro-benchmark variations as well as 40 real mobile game applications. For training sets we use the basic 36-benchmark training set (Figure 3.17.(a)) and combinations of the 36-benchmark set (Figure 3.17.(b)); for deployment sets we use another combinations of micro-benchmarks not used in the training sets (Figure 3.18.(a)) and the 40 real game set (Figure 3.18.(b)).

The 36-benchmark set (Figure 3.17.(a)) is derived from the micro-benchmarks described in Chapter 2: mb-verM, mb-TexM, mb-VerSh, mb-FragSh and mb-App. We use two observations to select the combinations of these parameters: 1) The effects of vertex memory (mb-VerM) on FPS, power and energy per frame were similar to those of vertex shader (mb-VerSh) and fragment shader (mb-FragSh). 2) The effects of vertex memory (mb-VerM) on the same metrics were also similar to those of texture memory (mb-TexM) except cases for extreme texture memory usage. Therefore we swept the workloads (from 0 to 5 by increasing the corresponding workload) of mb-VerM (for GPU workloads) and mb-App (for CPU workloads) respectively with fixed values for mb-TexM, mb-VerSh, and mbFragSh.

CPU \ GPU Cost Index	0	1	2	3	4	5
0	mb000	mb010	mb020	mb030	mb040	mb050/150/250
1	mb100	mb110	mb120	mb130	mb140/240	mb350
2	mb200	mb210	mb220	mb230/330	mb340	mb450
3	mb300	mb310	mb320	mb430	mb440	mb550
4	mb400	mb410	mb420	mb540		
5	mb500	mb510/520	mb530			

(a) The Basic 36-benchmark Training Set



(b) The Micro-benchmark Training Set

Figure 3.17: The Training Sets.

Moreover, the 36-benchmark set (Figure 3.17.(a)) is based on 4 categories of workloads (No, GPU, CPU, CGU dominant) as shown in Figure 3.3. We used cost indices of 2 and 3

CPU \ GPU Cost Index	0	1	2	3	4	5
0	mb000	mb010	mb020	mb030	mb040	mb050/150/250
1	mb100	mb110	mb120	mb130	mb140/240	mb350
2	mb200	mb210	mb220	mb230/330	mb340	mb450
3	mb300	mb310	mb320	mb430	mb440	mb550
4	mb400	mb410	mb420	mb540		
5	mb500	mb510/520	Mb530			

(a) The Micro-benchmark Deployment Set

CPU \ GPU Cost Index	0	1	2	3	4	5
0	AngryBirds	Csr racing skycastle	Bonsai bench MobileGPU mark	Frozen highway	seascape	Actionbike baseMark
1	Armored car	Deerhunter Farmville2	Anomaly2-nor.	Dream bike Epiccitadel Hercules	Anomaly2 high	3dmark_extream
2	hobbit	Terminate RealSteel Speedorz	300 Frontline Dday Shootemdown	3dmark_normal Dinosaur hunter		Frontline 2
3			Dinogun ship	Robocob Train Simulator	GPU bench	
4				EdgeofTomorrow 3D rating		
5	Assassin creed	Q3zomb 2/4 Mission berlin Truck Simulator	Real driving			40 App

(b) The Real-Game Deployment Set

Figure 3.18: The Deployment Sets.

to categorize the four different types of graphics workloads in this platform. However, each dominance area also could have different workloads, which will require different CPU and GPU frequency settings. Therefore, as shown in the sample of the training set (Figure 3.7), we further categorized each dominance area into nine additional workloads (benchmarks), corresponding to Low, Medium, and High workloads in terms of the normalized CPU and GPU cost. This basically requires 36 applications (4 categories x 9 benchmarks) for the training set. However, our observations show that benchmarks having a normalized CPU and GPU cost over 80 rarely exist and these entries are shaded in dark gray in Figure 3.17 and Figure 3.18. Note that each benchmark of the 36-benchmark training set has almost constant CPU and GPU workloads with low workload variation during runtime, allowing us to generate combined training sets (such as Figure 3.17.(b) and Figure 3.18.(a)) that are composed of a few or several workloads out of the 36-benchmark workloads to generate more dynamic workloads, which in turn can simulate various types of graphics workloads similar to real games. For example, as shown in Figure 3.17.(b), the low-variation benchmarks for the refinement step are simulated like the small rectangles (covering 3-4 entries) and the high-variation benchmarks are simulated like the long diagonal arrows or the big rectangles (covering several entries).

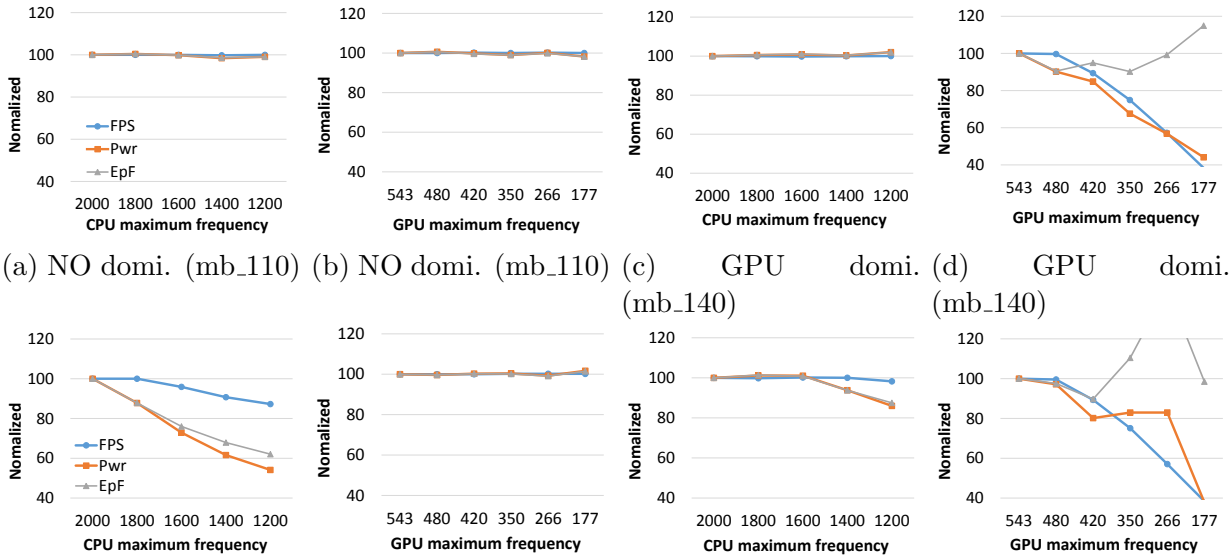
The deployment sets are composed of the combinations of the 36-benchmark set (Figure 3.18.(a)) and 40 real-game applications (Figure 3.18.(b)) derived from: previously published papers, traditional graphics benchmarks for performance comparison of commercial

products such as Anomaly2 or 3D mark, popular Android games like Angry Birds and Farmville, and games of popular game engine companies such as Unity or Gameloft's unreal engine. Of course any other games can be tested additionally; we also note that the No-dominant area has more games compared to other areas because many popular games were located in this area on ODROID-XU3 which is deploying high-performance CPUs and GPUs as we described in the start of this chapter.

For measurements and comparison: after capturing all data of Table 3.1 in every epoch, we compared power consumption of CPU-bc, CPU-lc, GPU, and memory. The average power consumption of CPU-lc in the training set was only 8.6% of the CPU-bc, CPU-lc, and GPU power and has no significant difference for CPU-bc and GPU frequency capping. Therefore, in this work we only consider CPU-bc and GPU frequency capping with the default maximum frequency(i.e., 1600Mhz) as the saturated frequency for CPU-lc; and we use the summation of CPU-bc, CPU-lc and GPU power consumption for a total power consumption.

In order to achieve a fair comparison with the baseline, we designed non-random micro-benchmarks and made efforts to find games that have very similar graphics workloads in every execution. In particular, the traditional benchmarks for graphics performance comparison had exactly same workloads in each execution. Moreover, to measure large sets of combined benchmarks and mobile games automatically and quantitatively, there is a need for an automated framework to run benchmarks, capture data and ensure repeatability of experiments. Towards this end, we developed an automatic measurement tool (AMT) for micro-benchmarks and mobile games using Linux shell scripts, Python modules, and XML files.

We then compare our proposed Co-Cap manager, which is implemented within the Linux kernel layer, with the default CPU and GPU governors (i.e., Interactive CPU governor and ARM's Mali Midgard GPU governor) using FPS, power (CPU-bc, CPU-lc, and GPU) and



(a) NO domi. (mb_110) (b) NO domi. (mb_110) (c) GPU domi. (mb_140) (d) GPU domi. (mb_140)

(e) CPU domi. (mb_410) (f) CPU domi. (mb_410) (g) CGU domi. (mb_440) (h) CGU domi. (mb_440)

Figure 3.19: FPS, Power and EpF Results of Different Types of Graphics Workloads.

EpF. To minimize variance across measurements, we perform repeated executions per application to get the average results.

Overhead and Epoch of our Manager: In order to evaluate performance and power consumption overhead of our manager, we measured the execution time of data capturing (3-4us) and Co-Cap management (1-3us) source codes. The total overhead (4-7us) time can be totally negligible compared to the default CPU governor epoch (50 ms) and GPU governor epoch (100 ms) in terms of performance (FPS degradation). Moreover, any noticeable power increase was not observed in terms of average power consumption when we add the data capturing and the Co-Cap management functions. In addition to the negligible overhead in terms of execution time and power consumption, we use the same epoch (i.e., 100 ms) with that of GPU governor because of two reasons: 1) We can easily solve the synchronization problem with the underlying CPU and GPU governors 2) A longer epoch compared to the default may affect performance degradation because of delayed frequency adaptation.

3.5.2 Experimental Results

3.5.2.1 Different types of graphics workloads

Figure 3.19 shows the effects of CPU (or GPU) maximum frequency capping on FPS, power, and EpF for some examples of the 36-benchmark training set. For illustration, we show a typical example benchmark from each different graphics workload (i.e., No-domi, GPU-domi, CPU-domi and CGU-domi) that provides specific examples for the general schemes of graph pattern analysis shown in Figure 3.8. (To clarify further, the average FPS and the power of the baseline are like these: mb110 (FPS: 60, Power: 864mW), mb140 (FPS: 58, Power: 1542mW), mb410 (FPS: 60, Power: 1765mW), mb440 (FPS: 58, Power: 1992mW).

3.5.2.2 Results of the Training Set

Figure 3.20 shows CPU-bc (CPU) and GPU energy savings per frame and FPS degradation compared to the default (i.e., Interactive CPU governor and ARM's Mali Midgard GPU governor) using the high-variation training set. Our results using Co-Cap show a significant combined CPU and GPU average energy savings of 8.9% (up to 18.3%) with insignificant FPS degradation of 0.9% across all the benchmarks. (For each characterized graphics workload, EpF improvement of 4.1%, 3.5%, 16.6%, 9.2% and FPS decline by 0.0%, 0.8%, 1.3%, 1.5% in No-, GPU-, CPU-, CGU-dominant benchmarks respectively). On average, the CPU's contribution to the energy savings is 7.8%, while the GPU's energy savings contribution is 1.1%. However, the results are totally different for GPU-dominant benchmarks (the second category in Figure 3.20 and 3.21); the GPU's contribution to the energy saving is 2.3% while the CPU's contribution is 1.1% of the 3.5% total energy savings. For CPU-dominant benchmarks (the third category in Figure 3.20 and 3.21), we observe 16.6% CPU and 0.3% GPU contributions. Based on our comprehensive experiments and analyses, we observe that the energy savings for mobile gaming benchmarks are mainly dependent on characteristics of the benchmarks (i.e., GPU energy savings from GPU-dominant and CPU energy savings from CPU-dominant benchmarks).

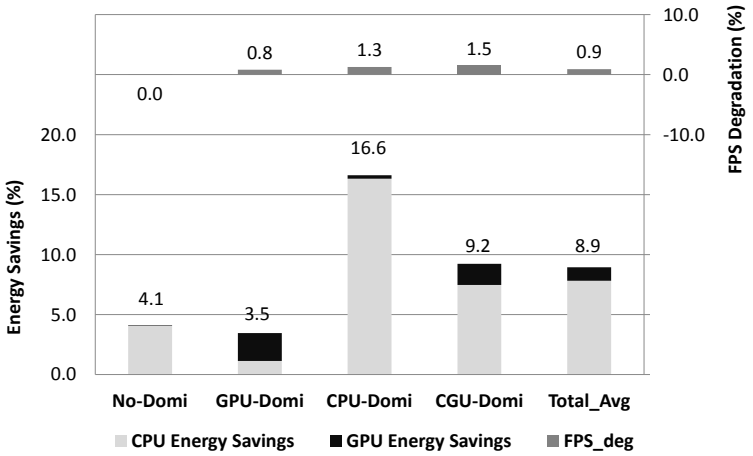


Figure 3.20: The Average Results of the Training Set (High-variation).

Figure 3.21 shows each result of the 60-benchmark training set. It is observed that the EpF is improved in most benchmarks except a few No-dominant benchmarks and the FPS is successfully maintained at almost similar to the baseline for all benchmarks. (Some No-dominant benchmarks are so lightweight that they could not have additional power reduction in spite of the minimum CPU and GPU frequency).

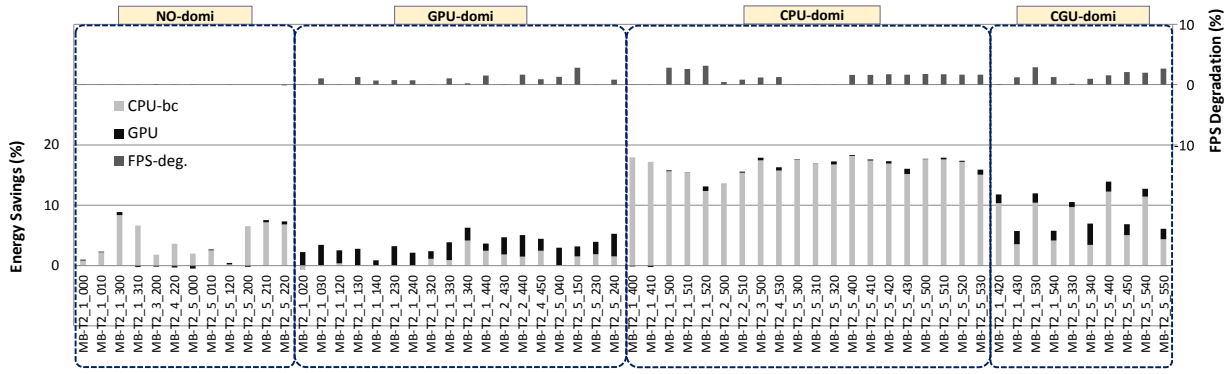
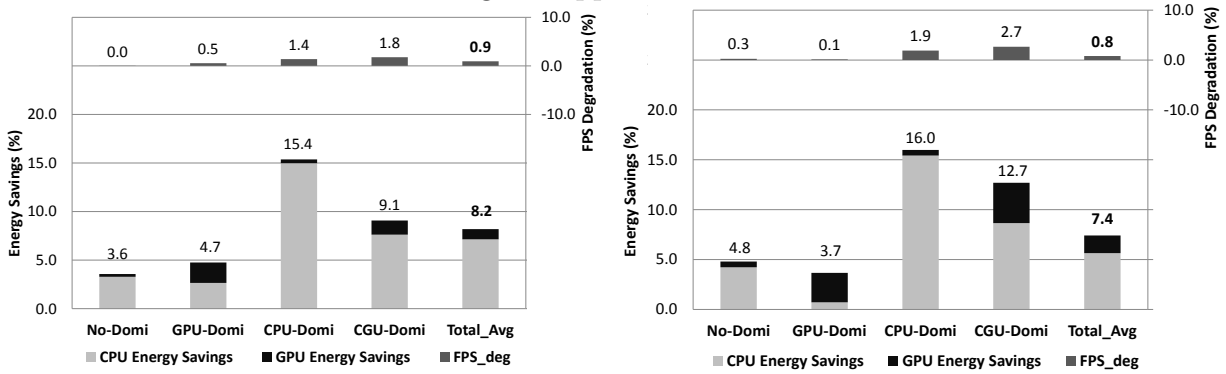


Figure 3.21: The Detailed Results of the Training Set (High-variation).

3.5.2.3 Results of the Deployment Sets

Figure 3.22 shows CPU-bc (CPU) and GPU energy savings per frame and FPS degradation compared to the default using the deployment sets: one high-variation benchmark set (not used in the training sets) and the other real-game application set respectively. (Because the general result patterns between micro-benchmark and real-game set are almost similar, we summarize the real-game set results within parentheses in the ensuing results). Our deploy-

ment set results also show a significant combined CPU and GPU average energy savings of 8.4% (7.4%) on average and up to 21.4% (27.6%) with insignificant FPS degradation of 0.9% (0.8%) across all the benchmarks. (For each characterized graphics workload, EpF improvement of 3.6% (4.8%), 4.7% (3.7%), 15.4% (16.0%), 9.1% (12.7%) and FPS decline by 0.0% (0.3%), 0.5% (0.1%), 1.4% (1.9%), 1.8% (2.7%) in No-, GPU-, CPU-, CGU-dominant benchmarks respectively). Especially, overall result patterns (and also even values) in the deployment sets are similar to those of the training set. These results clearly show that Co-Cap is able to achieve significant improvement in EpF with little FPS decline for all types of graphics workloads in the training and the deployment sets including real-game applications; especially combinations of micro-benchmarks can be utilized as training sets in mobile gaming workloads to resolve the challenges of finding various types of real-games and modification of source codes in real-game applications to fix or change graphics workloads.



(a) A 60 Micro-benchmark Set

(b) A 40 Real-Game Set

Figure 3.22: Average Results of the Deployment Sets

Additionally, Figure 3.23 and 3.24 show the results of each application in the deployment sets. Overall result patterns in each characterized graphics workload (i.e., No-, GPU-, CPU-, CGU-dominant workloads) are similar between benchmark and real-game applications. However, the variation of real-game applications in the same category is higher than that of the micro-benchmark set.

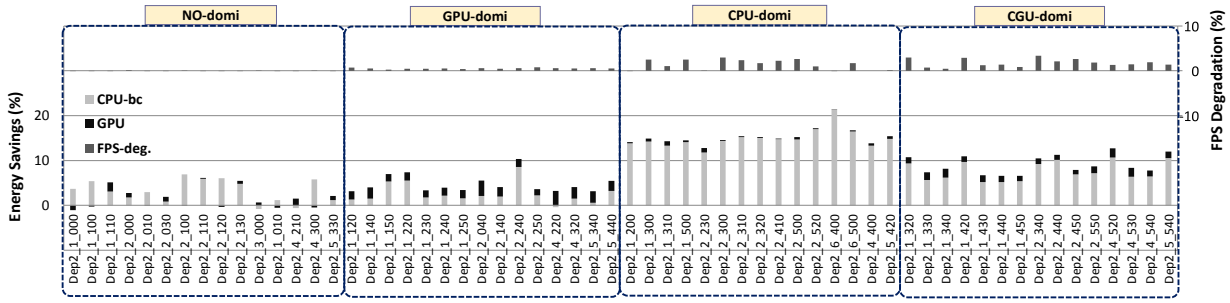


Figure 3.23: The results of the Deployment Set (MB).

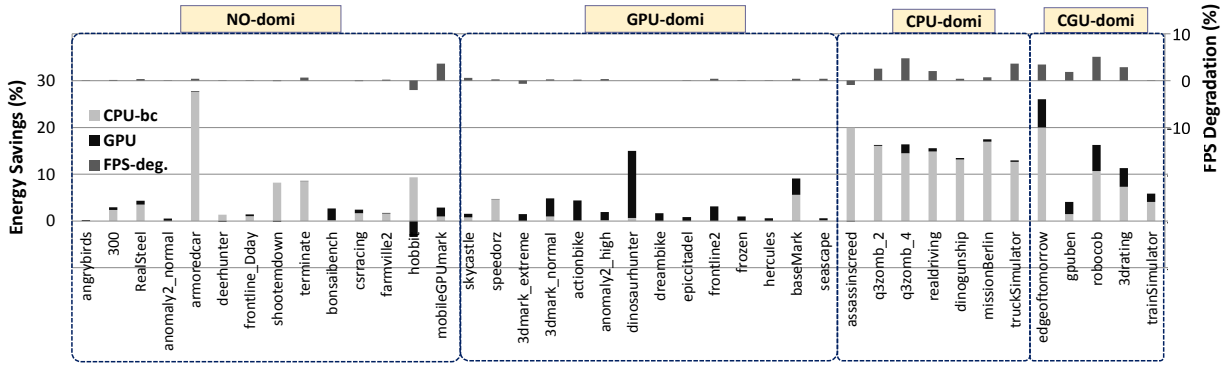


Figure 3.24: The results of the Deployment Set (RG).

3.5.3 Analysis and Discussion

Systematic Methodology for Completeness: Compared to the real-game based earlier work [77], we systematically establish completeness using the micro-benchmark training sets and the fine-grained refinement steps. First, with the help of our custom micro-benchmarks, now it is possible to generate appropriate CPU-GPU workloads for each entry, while it was very difficult to find games for some specific entries that are vacant. Therefore, while the saturated frequencies for the vacant entries were empirically speculated by using interpolated values of adjacent entries in our earlier work, now we can configure each saturated frequency systematically at each entry using the benchmark sets covering all entries. Second, while the refinement step in our earlier work was established empirically through just repeated overall Co-Cap evaluation for applications in the training set, the procedure of our fine-grained refinement steps (Figure 3.12) provides a systematic methodology using a greedy heuristic

with two different types of complete training sets based on the custom micro-benchmarks.

CPU and GPU Energy Savings: Through the results and analyses of the multiple training and deployment sets, EpF improvements differ based on the types of graphics workloads. We do observe that the energy savings for mobile gaming benchmarks are mainly dependent on characteristics of the benchmarks (i.e., GPU energy savings from GPU-dominant and CPU energy savings from CPU-dominant benchmarks) and that CPU dominant workload applications have better EpF improvement compared to GPU dominant workloads. Note that the default CPU governor supports cluster-based DVFS. We speculate that the main rendering process of graphics applications on Android system is executed on one single core even though there are four big CPU cores and four little CPU cores; moreover, the default interactive CPU governor scale up immediately to the maximum frequency at a certain state for performance improvements. Therefore, if we use a slightly lower maximum frequency removing frequency over-provisioning compared to the default maximum frequency, we can easily get significant power reduction with little FPS loss for CPU dominant applications. However, for GPU dominant workloads, we observe that an accelerator-based GPU is especially dedicated for rendering tasks and that the default GPU governor is conservative compared to the CPU governor in order to escape prospective rendering glitches. Therefore, the power reduction rate of GPU dominant workload (Figure 3.19.(d)) is less than that of the CPU dominant workload (Figure 3.19.(e)) for the same amount of FPS degradation. For No-dominant workloads, the minimum frequency of CPU-bc or GPU is still quite high for lightweight graphics application like Angrybirds as shown in Figure 3.24. For these kinds of ultra lightweight graphics workloads, we speculate that CPU-lc frequency capping in addition to CPU-bc and/or CPU power gating are potentials for additional energy savings within minimal FPS degradation on high performance HMPSoCs.

Consideration for Memory Intensive Workloads: The impacts of memory maximum frequency capping on FPS, Power and EpF are described in the Co-Cap Technical Report [76]. However, according to our observations, opportunities for energy savings from

the memory frequency capping do not exist. Therefore, an alternative considering memory intensive workloads is to use CPU or GPU governor. To do this, there could be two possible approaches: 1) One is to train/evaluate our LUTs including memory intensive applications and 2) the other is to build an additional model considering memory workloads in CPU or GPU governor with extra memory workload characterization. In this work, we choose the former approach because one of the main objectives is to provide a simple and easily portable methodology for rapidly changing platforms; moreover, we evaluate our manager including memory intensive applications which use the memory utilization (MU) from 17% to 98% in the deployment set, and achieved significant energy savings for the memory intensive applications: For example, 26% for edgeoftomorrow (MU = 70), 16% for robocop (MU = 83) and 9% energy savings for baseMark (MU = 76). And, our another work presents the latter approach and summarize as follows.

Memory-aware Cooperative CPU-GPU DVFS Governor for Mobile Games [46]: While DVFS techniques have been exploited previously for dynamic power management, contemporary techniques do not fully exploit the memory access footprint [59] [28] [49] [52] for graphics-intensive gaming applications, missing opportunities for energy efficiency. In this work, we for the first time propose a memory-aware cooperative CPU-GPU DVFS governor that considers both the memory access footprint as well as the CPU/GPU frequency to improve energy efficiency of high-end mobile game workloads. Our experimental results show that our proposed game governor achieves on average 13% and 5% improvement of energy efficiency with minor degradation of performance compared to default governors and state-of-the-art game governors.

Our proposed Co-Cap methodology shows promise for improving energy efficiency across a wide range of micro-benchmarks and mobile games, and also in being rapidly applicable for newer platforms as they emerge. However, our initial efforts still need to address a few open issues, such as: Can we use more sophisticated methods (e.g., statistical models or offline/online machine learning algorithms) to augment our existing heuristics for saturated

frequencies? How do we incorporate the effects of user inputs, which vary dynamically during game execution? These and other extensions will be our future work.

3.6 Conclusion

In this chapter, we proposed *Co-Cap, an energy-efficient CPU-GPU dynamic maximum frequency capping technique*. In the training phase, we first dynamically captured data such as utilization and frequency, estimated graphics workloads using the normalized CPU and GPU cost, and then configured the saturated frequency of CPU and GPU in each cost window. We then evaluated the efficacy of Co-Cap using new sets of micro-benchmarks and games. Our experimental results using more than 200 micro-benchmarks and 40 real games on the ODROID platform show that Co-Cap improves energy per frame by 8.9% and 7.8% (16.6% and 15.7% in CPU dominant applications) on average and achieves little FPS decline by 0.9% and 0.85% (1.3% and 1.5% in CPU dominant applications) on average for the training- and deployment sets respectively, compared to the default CPU- and GPU governors, with negligible overhead in execution time and power consumption on ODROID-XU3. Our ongoing and future work includes proposing a smart CPU-GPU synergistic governor using more sophisticated methodology such as statistical models or machine learning algorithms. Finally, while this methodology was targeted mainly for mobile games, it can also be applicable for various types of CPU-GPU integrated graphics applications.

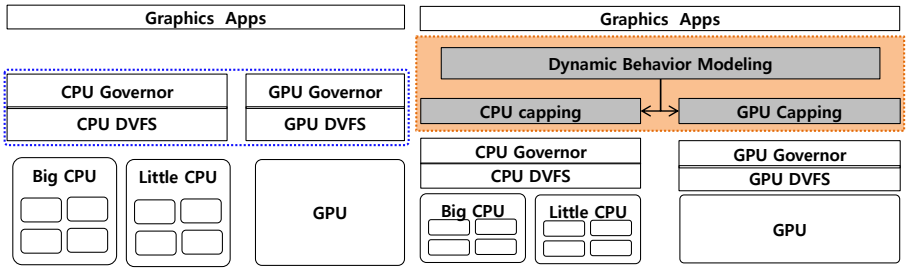
Chapter 4

Hierarchical FSM-based Integrated CPU-GPU Frequency Capping

4.1 Introduction

Mobile games are an increasingly important application workload for mobile devices. These games often demand high performance while rapidly depleting precious mobile battery resources, resulting in low energy efficiency. The recent trend towards Heterogeneous MultiProcessor Systems-on-Chip (HMPSoC) architectures (e.g., ARM's big.LITTLE with integrated GPU) attempt to meet the performance needs of mobile devices, and rely on software governors for power management in the face of high performance. Contemporary commercial mobile platforms (e.g., SAMSUNG Galaxy S6 and NEXUS 6) deploy separate governors for CPU and GPU DVFS power management (Figure 4.1(a)), but miss opportunities to coordinate power management between the CPU and GPU domains for improved energy efficiency. Some recent research efforts have proposed integrated CPU-GPU DVFS policies [80] [51] for a small set of mobile games, assuming a fairly static workload behavior

(dotted box in Figure 4.1(a)). However, mobile games generally exhibit highly dynamic behaviors through a varying mix of CPU and GPU workloads; existing software governors are unable to exploit this behavioral dynamism to further improve energy efficiency while delivering acceptable user experience for mobile gaming. To address these issues, we present HiCAP (Figure 4.1(b)): a hierarchical FSM (HFSM) based integrated CPU-GPU DVFS governor that: 1) naturally models the application's dynamic behavior using the HFSM model, and 2) uses a simple, cooperative CPU and GPU frequency capping technique to achieve improved energy efficiency.



(a) Contemporary Governors (b) HiCAP Governor

Figure 4.1: CPU-GPU Mobile Governors

Many mobile games have multiple levels of play with complex stages, and thus the games themselves are typically designed hierarchically [61] [87]. Furthermore, the class of hierarchical FSMs provide an excellent abstraction for behavioral modeling of many complex embedded systems [31] [69]. Therefore we believe HFSMs offer a natural and intuitive modeling abstraction for capturing the behavioral dynamism of a game, for applying effecting DVFS policies.

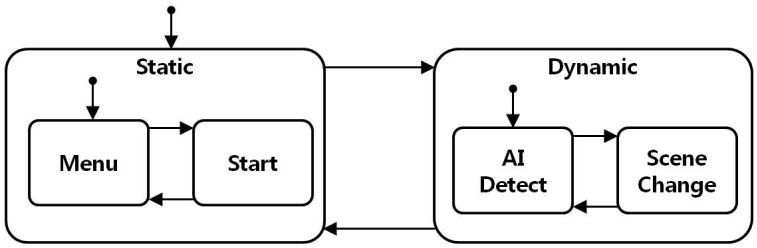
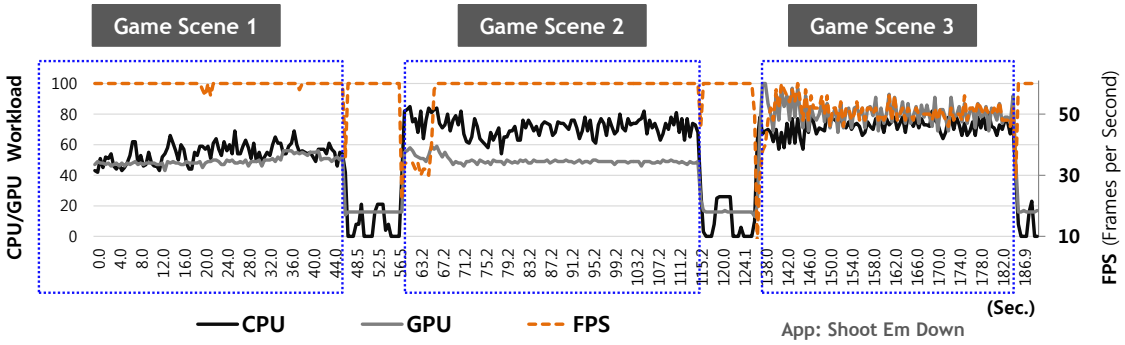


Figure 4.2: HFSMs in Game Design.

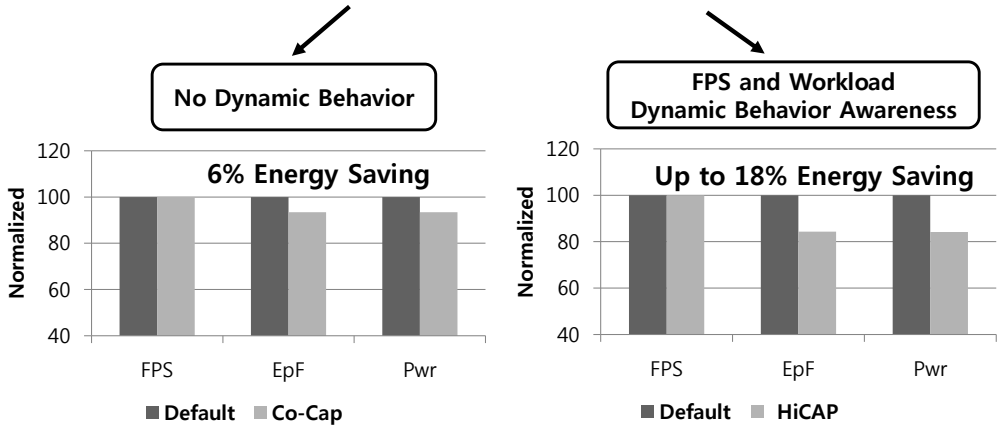
For instance, Figure 4.2 shows the top-level HFSM of a typical game that proceeds from the game set-up (*Static* superstate) to game execution (*Dynamic* superstate); of course dur-

ing execution the *Dynamic* superstate will contain a complex hierarchy of game levels, stages and conditions to support various features of individual games. This motivates the use of HFSMs in our HiCAP governor model.

Now consider Figure 4.3, where use a sample game *ShootEmDown* to motivate HiCAP's need for HFSM modeling, and HiCAP's opportunity to improve energy efficiency by exploiting game dynamism. Figure 4.3(a) captures the dynamism for different game scenes/actions, showing the delivered Fram-es-per-Second (FPS) (dotted orange) [1], and the CPU (black) and GPU (gray) workloads, normalized to their respective maximums. This figure clearly shows the need for HFSMs to capture the game's behavior hierarchically: *Game Scene 1* is fairly static, whereas *Game Scenes 2* and *3* are highly dynamic, but with varying FPS, CPU and GPU workloads.



(a) Different Types of Game Dynamism



(b) No Dynamic Behavior (c) Dynamic Behavior

Figure 4.3: Sample Mobile Game (ShootEmDown)

[1]Note that the delivered Frames-per-Second (FPS) is a typical metric for gaming user experience, with a maximum FPS of 60 in the Android system.

Figures 4.3(b) and (c) highlight HiCAP's opportunity for energy savings by exploiting this game's dynamic behaviors. When the game's FPS and CPU/GPU workload dynamism are not captured (Figure 4.3(b)), our previous Co-Cap technique [77] achieves a modest (up to 6%) improvement. However, the game's execution exhibits various levels of dynamism, and our HiCAP governor is able to achieve significant (up to 18%) improvement in energy efficiency over state-of-the-art static modeling governors by exploiting the dynamism in the game's FPS and the CPU/GPU workload, clearly demonstrating the potential for improved energy efficiency.

This work makes the following specific contributions:

- We propose a Hierarchical FSM (HFSM) based dynamic behavior modeling strategy for mobile gaming
- We present HiCAP: a cooperative CPU-GPU governor that deploys a simple maximum frequency-capping methodology exploiting the HFSM for dynamic DVFS
- We present experimental results on a large set of 37 real mobile games with dynamic behaviors, showing significant energy savings of up to 18% in Energy-per-Frame (EpF) with minimal loss in FPS performance.

4.2 Related Work

With the emergence of high performance integrated mobile GPUs, several research efforts have proposed integrated CPU-GPU DVFS governors: Pathania *et al.*'s [82] integrated CPU-GPU DVFS algorithm didn't consider quantitative evaluation for energy savings (e.g., per-frame energy or FPS per watt); their next effort [80] further developed power-performance models to predict the impact of DVFS on mobile gaming workloads, but did not model the

games' dynamic behaviors; and Kadjo *et al.* [51] used a queuing model to capture CPU-GPU-Display interactions across a narrow range of games exhibiting limited diversity in CPU-GPU workloads.

In another direction, frequency capping techniques have been proposed as a simple yet effective strategy for DVFS. Li *et al.* [58] initially introduced frequency capping for energy efficiency, but their approach is restricted to only CPU governor. Most recently, we proposed a coordinated CPU-GPU maximum frequency capping technique [77] and applied it to a range of mobile graphics workloads, but assumed static characterization of each application, and did not consider dynamic behaviors for different CPU/GPU graphics workloads and the Quality of Service (QoS) requirement.

Hierarchical FSMs [31] were proposed as an effective semantic model to capture complex system behavior, with many variants such as Statecharts [44] providing modeling support for complex embedded systems [69]. Many embedded systems – particularly for streaming multimedia – have combined dataflow with HFSM models to enable tasks such as runtime resource allocation and dynamic task mapping (e.g., [50]).

To the best of our knowledge, our HiCAP approach is the first to introduce a hierarchical FSM-based dynamic behavior modeling strategy for mobile gaming considering QoS requirements; and the first to exploit the dynamics of CPU/GPU graphics workload variation through a simple maximum frequency capping strategy for energy efficiency on modern HMPSoC platforms by avoiding frequency over-provisioning while delivering acceptable user QoS.

4.3 Approach

4.3.1 Preliminaries

We begin by defining terminology used in this work.

CPU/GPU Workload: Mobile workload variations are typically quantified using a cost metric that is a product of the utilization and frequency [16] [82]. Accordingly, as shown in Equation(1), we deploy the normalized CPU and GPU costs [77].

$$Normalized\ Cost = \frac{Curr\_Util.\ \times\ Curr\_Freq.}{Max\_Util.\ \times\ Max\_Freq.} \tag{4.1}$$

CPU/GPU Cost Matrix and Quadrants: To model the dynamics of game behavior across the CPU and GPU domains, we develop a CPU/GPU Cost Matrix using the normalized CPU (y-axis) and GPU (x-axis) costs, where the dominance of that component (CPU- or GPU-) increases along each axis. Figure 4.4 shows our set of 37 gaming applications that populate different entries in this matrix, corresponding to the CPU/GPU intensity of its workload. This matrix has 4 major quadrants: No CPU-GPU dominant, CPU dominant, GPU dominant and CPU-GPU dominant workloads [77].

CPU \ GPU Cost	0–20	20–40	40–60	60–80	80–90	90–100
0–20		Dream Bike Sky Castle 2	Bonsai Bench	Extreme Motorbike		
20–40	Armored Car	Anomaly2 Low	Hercules Anomaly2 Normal	Epic Citadel	Anomaly2 High Implosion	
40–60	Fast and Furious 7	Madden NFL 15 Micro-bench 102	Frontline Dday Godus ShootEm Down S Micro-bench 110 Micro-bench 111	3D Mark - Normal		
60–80		Q3 Zombie Map 1 Terminator Genisys Micro-bench 100 Micro-bench 101	Micro-bench 112 Dino Gunship ShootEmDown A	ShootEmDown D Train Simulator		
80–90		Micro-bench 103	Micro-bench 113	RoboCop Edge of Tomorrow		
90–100	Assassin Creed : P	Turbo FAST (Net) Q3 Zombie Map 2 Q3 Zombie Map 4				A Total Set : 37

Figure 4.4: A set of benchmarks.

Frequency Over-Provisioning and Frequency Capping: Existing software governors often over-provision frequency, which provides energy-saving opportunities by simply capping the maximum operating frequency without loss of quality [77].

Saturated Frequency Look-up Tables: We can enter saturated (capped) frequencies (determined statically [77] or dynamically) in each entry of the CPU-GPU cost matrix, re-

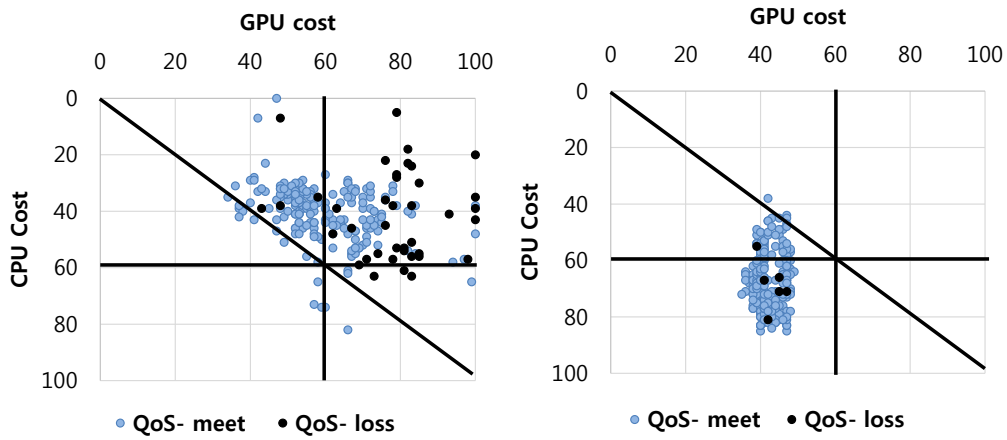
sulting in a CPU/GPU saturated frequency look-up table that covers the entire cost matrix. This saturated frequency look-up table can then be used for determining the new CPU and GPU maximum frequency settings by the runtime governor.

Having discussed preliminaries, we now describe HiCAP's approach for using HFSMs to model dynamic behavior for mobile gaming (Chapter 4.1.2), and HiCAP's heuristics for configuring CPU/GPU saturated frequencies through a runtime frequency-capping methodology (Chapter 4.1.3).

4.3.2 HFSM-based Dynamic Behavior Model

4.3.2.1 Game Dynamism Analysis

Although each game may be situated in a specific quadrant of the CPU/GPU cost matrix (Figure 4.4), during runtime the game will exhibit dynamic behavior that covers a dynamic footprint of the CPU/GPU cost matrix. For instance Figure 4.5 shows the dynamic footprints of two benchmarks (*3DMark-Normal* and *Dino-Gunship*) during a 500ms execution snapshot, with the blue dots representing states that satisfy QoS (labeled as *QoS-meet*) while the black dots represent states that are unable to meet the QoS requirement (labeled as *QoS-loss*). Furthermore, we also note that the CPU-GPU workload dynamism can be characterized as being CPU-dominant (*CPU-D*) or GPU-dominant (*GPU-D*), if the CPU/GPU workload can be characterized quantitatively. For the specific examples in Figure 4.5, we note that the footprint of the *3DMark-Normal* game (Figure 4.5(a)) is mainly located in the right-side of the diagonal line, which implies that GPU cost is higher than CPU cost (i.e., this is a GPU-dominant workload). Similarly, the game *Dino-Gunship* (Figure 4.5(b)) has the footprint of a CPU-dominant workload. In summary, using this strategy we can characterize each game's dynamic behavior.



(a) 3DMark-Normal (b) Dino-Gunship

Figure 4.5: Footprints of Game Dynamism

4.3.2.2 HFSM-based Modeling

We use hierarchical FSMs (HFSMs) to model each game's dynamic behavior, as shown in Figure 4.6. We use a formalism similar to StateCharts, where each state represents a specific dynamic behavioral state of a game application, and the state may be further refined into another lower-level FSM. We call the inside FSM the sub-state and the outside FSM the super-state in the StateCharts language [69].

Our HFSM model starts with two super-states at the highest level labeled *QoS-meet* and *QoS-loss*, and each super-state has two sub-states labeled *CPU-D* and *GPU-D* as shown in Figure 4.6. During design optimization, new states can be added heuristically to this model. For example, while meeting a QoS requirement, let's assume that two states are fluctuating continuously (e.g., repeated *QoS-meet* and *QoS-loss*) for a certain amount of time. In this situation, the output of each state also fluctuates continuously, resulting in performance or power overheads. To improve this situation, a new QoS-transient (*QoS-tr*) state can be designed, that generates a new output. Furthermore, in terms of CPU/GPU workload (cost) – i.e., multiplication of frequency and utilization – a continuous maximum utilization (*Cont-UMax*) may result in FPS degradation in the *QoS-loss* state. Because this may further degrade performance, we can design a new state to deal with this specific situation. Also note that due to the hierarchical structure of the HFSM, sub-states such

as *CPU-D*, *GPU-D*, or *Cont-UMax* are in turn also super-states at a lower level, consisting of sub-states themselves. Due to space limitations, detailed additional sub-states are not described here, but can be found in our Technical Report [79].

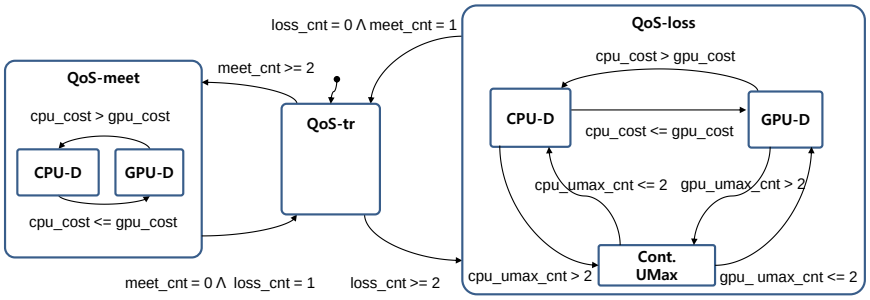


Figure 4.6: Hierarchical Finite State Machine

The HFSM in Figure 4.6 has switching conditions shown as annotations on the HFSM edges. We partition the execution time of applications into 500ms epochs [79]. At the start of each epoch, using FPS and workload data captured during the previous epoch, our HFSM determines whether to make the transition or not, and takes the appropriate action. The QoS-target is changeable by users, but for this study we use the maximum FPS (60 in Android system) to compare directly the default and related governors. We now provide a detailed description of our switching heuristic using Figure 4.6 and Algo. 1.

The HFSM is initialized to the *QoS-tr* super-state (the default state among the super-states). If the average-FPS of the previous epoch meets the QoS-target for the past two states ($meet\_cnt >= 2$), then the FSM transitions to the *QoS-meet* state (line 2). In this state, if the average-FPS loses the FPS-target for the first time, then the FSM transitions to the *QoS-tr* state (line 17), not directly to *QoS-loss* state. However, if the average-FPS loses FPS-target two times consecutively ($loss\_cnt >=2$), then the FSM transitions to the *QoS-loss* state (line 8). We use two counters ($meet\_cnt$ and $loss\_cnt$) to detect the *QoS-tr* state.

For sub-state transitions, the *QoS-loss* super-state can be further refined into *CPU-D*, *GPU-D*, and *Cont-UMax* sub-states according to various conditions: if CPU cost is larger than GPU cost, the *CPU-D* sub-state will be the new active state (line 9); if GPU cost is

bigger than or equal to GPU cost, the *GPU-D* sub-state will be the new active state (line 11); and if CPU/GPU maximum utilization counters (*cpu_umax_cnt* or *gpu_umax_cnt*) are bigger than specific thresholds, the *Cont-UMax* sub-state will be the new active state (line 14). In a hierarchical FSM model, same sub-states such as *CPU-D* and *GPU-D* can be located in different super-states (line 3 and 5).

Until now, we have not considered outputs (reactions) generated by our hierarchical FSM. Once the two transitions in super- and sub-state are triggered, possible reactions include the generation of events and/or assignments to variables. Using the HFSM model, we assign new appropriate CPU/GPU saturated frequencies as HFSM outputs, which are then applied in the next epoch.

4.3.3 Frequency-Capping

We adopt a CPU and GPU frequency capping heuristic for the following reasons: 1) Simplicity: adding a capping module on top of the default (or custom) CPU/GPU governors is easier than complicated integrated CPU/GPU governors. 2) Portability: capping is easily adaptable to newer platforms. 3) Efficacy: our previous work [77] has shown overall improvements for different types of graphics workloads. 4) Elimination of wasteful frequency over-provisioning: commercial CPU governors typically scale frequency using utilization based thresholds. In other words, if CPU utilization is greater than or equal to a certain threshold, the governor sets the highest frequency from the corresponding frequency [58]. This is also similar in GPU governor but more conservative. However, in both cases these approaches may result in frequency over-provisioning, since the frequency is set to a higher level even when the target QoS has been met at maximum utilization. This results in wasted power.

To configure appropriate saturated frequencies, we design dynamically changing CPU and GPU HiCAP (HiC) look-up tables (LUTs), which determine the new CPU and GPU maximum frequencies of cost quadrants for the next epoch. We label the upper-bounded

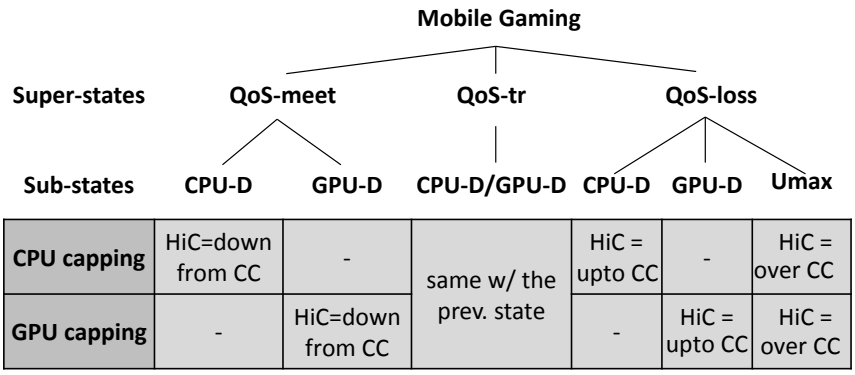


Figure 4.7: Different Capping Policies as Outputs

Algorithm 1 The HFSM Heuristic Pseudo-code

```
    Every epoch
 1: Calculate average FPS, CPU and GPU Costs per epoch

 2: if meet_cnt ≥ 2 then                                  ▷ QoS-meet
 3:     if cpu_cost > gpu_cost then                            ▷ CPU-D
 4:         HiC = CPU Freq-Cap down from CC
 5:     else                                                   ▷ GPU-D
 6:         HiC = GPU Freq-Cap down from CC
 7:     end if
 8: else if loss_cnt ≥ 2 then                             ▷ QoS-loss
 9:     if cpu_cost > gpu_cost then                            ▷ CPU-D
10:         HiC = CPU Freq-Cap up to CC
11:     else                                                   ▷ GPU-D
12:         HiC = GPU Freq-Cap up to CC
13:     end if
14:     if cpu (or gpu)_umax_cnt > 2 then                      ▷ UMax
15:         HiC = CPU or GPU Freq-Cap over CC
16:     endif
17: else                                                  ▷ QoS-tr
18:     Same Frequency-Cap with the previous state
19: end if
20: Newely updated CPU and GPU HiC LUTs for next epoch
```

frequencies from Co-Cap [77] as CC. By exploiting the hierarchical property of HFSMs (Figure 4.7), we apply different capping policies in each sub-state. For example, the capping policies between *QoS-meet* *QoS-tr*, and *QoS-loss* state are fundamentally different. In the *QoS-meet* state, lower saturated frequencies than CC are set because QoS-target is already met (line 4 and 6). However, in the *QoS-loss* state, almost similar frequencies to CC are required to have competitive performance (line 10 and 12). In particular, for continuous maximum utilization in *QoS-loss* state, temporarily higher frequency than CC is used to prevent FPS reduction (line 15). In the *QoS-tr* state, the same saturated frequencies of the previous state (line 18) are used to reduce fluctuation of outputs (i.e., overheads). According to the characterization of different types of graphics workloads (i.e., for CPU (GPU)

dominant workloads, energy saving within minimal FPS decline mainly results from CPU (GPU) capping), we mainly scale CPU (GPU) saturated frequency for *CPU-D* (*GPU-D*) sub-states.

4.4 Experimental Results

4.4.1 Experimental Setup

We evaluated our HiCAP governor on the ODROID-XU3 development board installed with Android 4.4.2 and Linux 3.10.9; Table 4.1 summarizes our platform configuration. The platform is equipped with four TI INA231 power sensors measuring the power consumption of big CPU cluster (CPU-bc), little CPU cluster (CPU-lc), GPU and memory respectively. The CPU supports cluster-based DVFS at nine frequency levels (from 1.2Ghz to 2.0Ghz) in CPU-bc and at seven frequency levels (from 1.0Ghz to 1.6Ghz) in CPU-lc, and GPU supports six frequency levels (from 177Mhz to 543Mhz).

Table 4.1: Platform Configuration

Feature	Description
Device	ODROID-XU3
SoC	Samsung Exynos5422
CPU	Cortex-A15 2.0Ghz and Cortex-A7 Octa-core CPUs
GPU	Mali-T628 MP6, 543Mhz
System RAM	2Gbyte LPDDR3 RAM at 933MHz
Mem. Bandwidth	up to 14.9GB/s
OS(Platform)	Android 4.4.2
Linux Kernel	3.10.9

Benchmark Set: As shown in Figure 4.4, we use 37 gaming applications covering different types of graphics workloads; many are derived from previously published papers ([75] [82] [80] [51] [77]).

For comparison: We compare our HiCAP governor against the default and two state-of-the-art related governors ([80] and [77]). The default corresponds to the independent CPU and GPU governors in Linux (Interactive CPU governor and ARM's Mali Midgard GPU governor). The first state-of-the-art governor [80] (represented as PAT15) proposed an in-

tegrated CPU-GPU DVFS strategy by developing predictive regression power-performance models. The second state-of-the-art governor [77] (represented as Co-Cap16) presented a coordinated CPU and GPU maximum frequency capping technique using upper-bounded saturated frequencies configured in a static training phase.

Overhead: Our power manager was implemented in kernel layer on top of CPU and GPU governors. The execution time of our manager per epoch is within 10us, which is totally negligible compared to the epoch period (500ms). Moreover, we did not observe any noticeable increase in average power consumption due to the power manager.

4.4.2 Automatic Measurement Tool

To measure a large set of mobile games automatically and quantitatively, there is a need for an automated framework to run games, capture key execution characteristics and ensure repeatability of experiments. Towards this end, we developed an automatic measurement tool for mobile games (AMTG) using Linux shell scripts, Python modules, and XML files. Figure 4.8 shows the hardware experimental setup, and all AMTG modules (Figure 4.9) are executed on the Linux-host with USB connection to the device (ODROID-XU3).

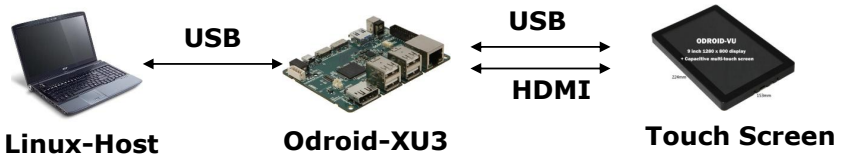


Figure 4.8: Experimental Setup.

Using this tool, we can capture the average values of FPS, total power (CPU-bc, CPU-lc, and GPU) and energy per frame (EpF) with repeated runs of each benchmark from the mobile device for a certain amount of time, and average their measurements.

The AMTG is based on the monkeyrunner tool [8] and composed of four major components: 1) Measurement Setup and Execution of monkeyrunner (Linux shell-scripts). 2) Game Manifestation (XML). 3) Execution of AMT Python Module. 4) Output files, as

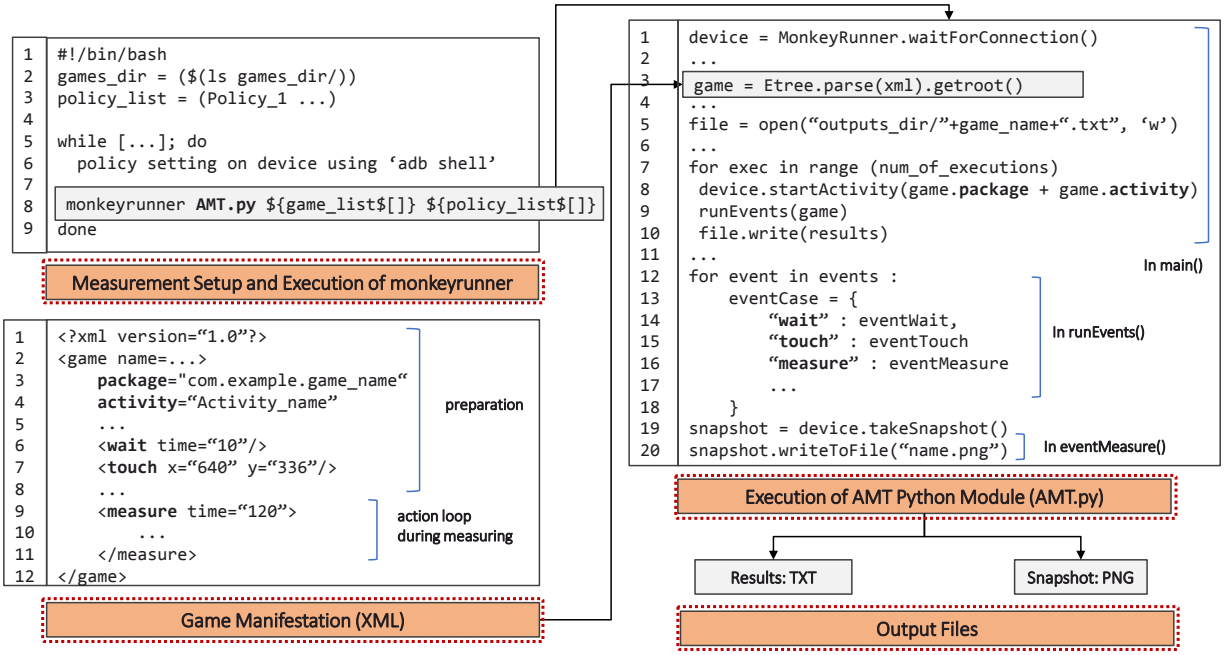


Figure 4.9: Automatic Measurement Tool.

shown in Figure 4.9. The monkeyrunner tool provides an API (*MonkeyRunner.waitForConnection()*) for writing programs that control an Android device or emulator from outside of Android code. With monkeyrunner, we write a Python program that runs an installed Android application, sends input values to it, takes screenshots of its user interface, and stores screenshots on the host. Pseudo-codes in each module will describe main functions for each component.

Measurement Setup and Execution of monkeyrunner: In order to decide a directory which has games for measurements and candidate policies, a measurement setup process is required. Therefore, a specific directory and policies for comparison should be defined in this stage. For example, as shown in the pseudo code of Figure 4.9 (top-left), *games_dir* (Line 2) and *policy_list* (Line 3) were defined. After setting each candidate policy (Line 6), the monkeyrunner program is repeatedly executed with the three arguments (a Python module name, a game to measure and the corresponding policy) (Line 8).

Game Manifestation: Before describing the Python module to run each game on a device, the manifestation of package name, activity name, and events should be defined in

advance. This information is specified in an XML file (one XML file per game). The XML file (down-left) is composed of two parts: a preparation part and an action part. In the preparation part, the names of package and activity (Line 3 and 4), all positions for menu settings after loading times (Line 6 and 7) should be defined manually because each application has different package and activity names, settings and loading times. In the action part, we define the measuring time (e.g., 120 seconds in this work) (Line 9) and can optionally add additional workloads periodically using touch, press or/and drag events in Line 10.

Execution of AMT Python Module: The Python module (AMT.py, top-right) connects the current device (ODROID-XU3) and returns a *MonkeyDevice* object (Line 1). After parsing the corresponding XML file (Line 3), it runs the package and activity of the game (Line 8) and executes the input events (Line 9), which are defined in the game manifestation file and are implemented for each event in this Python module (Line 12-18). Finally according to change of the *num_of_executions*, the number of measurements for each policy in each game will be defined.

Output Files: Output files are defined in the AMT Python module (Line 5 and 20). We generate two different types of output files: txt-based result files, png-based snapshot files. Each opened TXT file (Line 5) will be written by results (Line 10). The result files include the average FPS and power data during the measuring time in addition to all captured data such as CPU/GPU utilization, frequency and cost function etc; they can be obtained using '*adb shell dmesg*' and '*grep*' commands. The snapshot files have the start- and the finish-screenshot of the measuring time of each application, and can be used to evaluate correctness of each execution.

4.4.3 Results and Analysis

Figure 4.10 summarizes the average results of the benchmark in FPS and EpF. Our HiCAP manager improves energy per frame by 18%, 14% and 8% on average compared to the default,

PAT15, and Co-Cap16 respectively, with negligible FPS decline of 1.5% and 1.2% on average compared to the default and Co-Cap16 respectively, and 2.9% better FPS than PAT15.

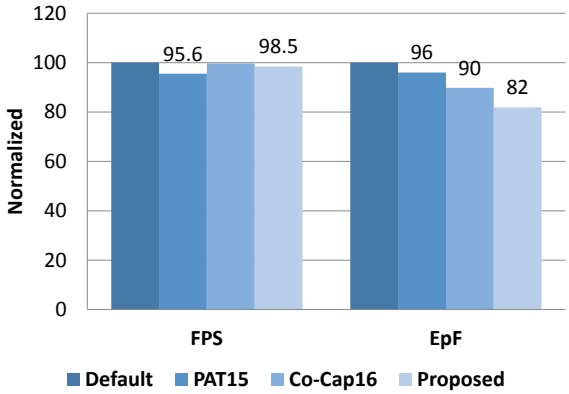


Figure 4.10: Average Results of the Benchmark Set

Comparison to Co-Cap16 [77]: Figure 4.11 shows CPU-bc (CPU) and GPU energy savings per frame and FPS degradation compared to Co-Cap16. (Comparisons to the default are detailed in our Technical Report [79] and omitted here due to lack of space; but we summarize these results within parentheses in the ensuing analysis). Our results using HiCAP show a significant combined CPU and GPU average energy savings of 8% (18% compared to the default) with insignificant FPS degradation of 1.2% (1.5%) across all the benchmarks. On average, the CPU's contribution to the energy savings is 6.5% (15%), while the GPU's energy savings contribution is 1.5% (3%).

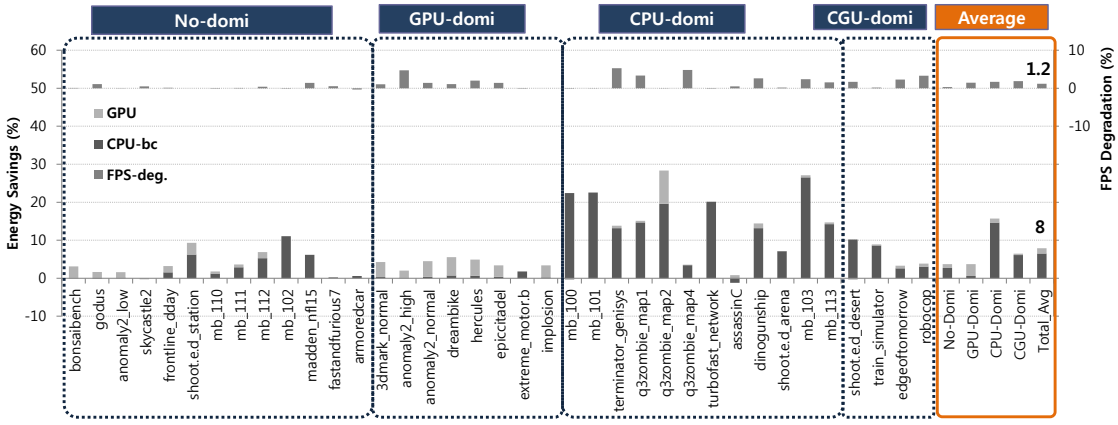


Figure 4.11: FPS and Energy Savings Comparison of Co-Cap16 [77] vs. our HiCAP

However, the results are totally different for GPU-dominant benchmarks (the second category in Figure 4.11); the GPU's contribution to the energy saving is 3% (7%) while the

CPU's contribution is 1% (1%) of the 4% (8%) total energy savings. For CPU-dominant benchmarks (the third category in Figure 4.11), we observe 15% (29%) CPU and 1% (1%) GPU contributions.

Our results clearly show that HiCAP achieves significant improvements in energy efficiency over the default governor and Co-Cap16 [77]; and that these are fair comparisons, since we used the same ODROID-XU3 platform and similar games in our experiments. We also note that the results shown in [51] (using the Intel Baytrail SoC platform and executing a small set of No- and GPU-dominant benchmarks) are very different from ours: their work showed a larger savings from the GPU (13.3%) than the CPU (4.1%), because the GPU typically consumes more energy than the CPU during graphics applications. However unlike our HiCAP work, their experiments did not comprehensively cover a wide range of mobile games exhibiting dynamism. Indeed based on our comprehensive experiments and analyses, we observe that the energy savings for mobile gaming benchmarks are mainly dependent on characteristics of the benchmarks (i.e., GPU energy savings from GPU-dominant and CPU energy savings from CPU-dominant benchmarks) and platform characters such as CPU/GPU minimum/maximum frequency and the number of CPU/GPU frequency levels, in addition to the governor algorithm itself.

Comparison to PAT15 [80]: As shown in Figure 4.12, our HiCAP governor shows a significant average energy savings of 14%, with a concomitant FPS improvement of 2.9% across all the benchmarks compared to PAT15. Note that their work proposed linear regression-based prediction models and evaluated a small set of mobile games that have specific workload characteristics (mainly using CPU-dominant benchmarks).

When we compared their approach on a larger set of varying graphics benchmarks, the performance (FPS) prediction was up to 20% worse than the default for some GPU-dominant benchmarks such as *Anomaly2 high* and *Ano- maly2 normal* (the second category in Figure 4.12). In addition to the CPU-GPU frequency under-provisioning for some benchmarks with FPS degradation, their governor also allocates over-provisioned CPU/GPU frequencies

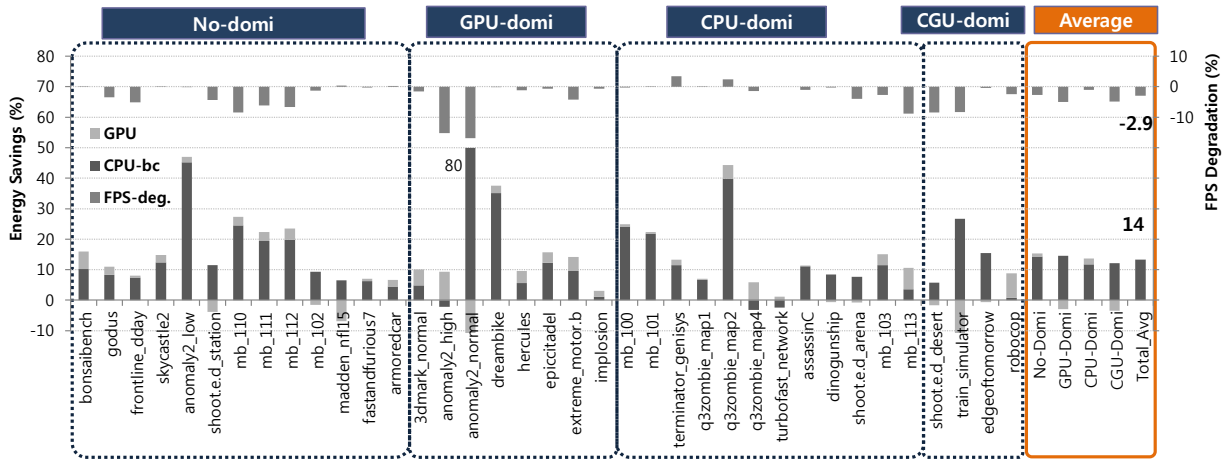


Figure 4.12: FPS and Energy Savings Comparison of PAT15 [80] vs. our HiCAP

for several benchmarks such as *Anomaly2 low* and *dreambike* which have no FPS degradation but worse energy savings than our HiCAP governor. Note that their approach also had average energy savings of 16% with FPS degradation of 3% across CPU-dominant benchmarks compared to the default, but had 2% and 8% worse average energy savings with FPS degradation of 3% and 7% for No- and GPU-dominant benchmarks respectively. Based on these results analysis, we observe that characteristics of gaming benchmarks (CPU- or/and GPU-dominant and Low/High workloads on the evaluated platform) are very important factors and results of each proposal also are dependent on the characteristics of benchmarks.

In summary, our HiCAP HFSM-based dynamic behavior model for mobile gaming is able to detect a specific state in terms of FPS-target and CPU/GPU workload; thus HiCAP is clearly able to eliminate CPU/GPU frequency over-provisioning using a simple frequency capping technique. Therefore, our HiCAP manager was able to achieve better results in energy saving, with minimal FPS degradation (and sometimes better FPS) compared to the default governor, as well as the most recent research efforts [80] [77].

4.5 Conclusion

In this chapter, we proposed *HiCAP: a Hierarchical FSM-based Dynamic Integrated CPU-GPU Frequency Capping Governor for Energy-Efficient Mobile Gaming*. HiCAP exploits the inherent hierarchical behavior of mobile games through an HFSM modeling approach; and uses a simple-yet-highly-effective capping technique to eliminate wasteful frequency over provisioning present in previous governors. We analyzed a large set of 37 mobile games exhibiting dynamic behaviors and developed a hierarchical FSM model that captures the games' dynamism. We then configured the CPU/GPU saturated frequency at run-time using a simple frequency-capping methodology as outputs of the HFSM state transitions. Our experimental results on the ODROID-XU3 platform across these 37 mobile games show that our HiCAP governor improves energy per frame by 18%, 14% and 8% on average compared to the default, PAT15 [80], and Co-Cap16 [77] governors respectively, with little FPS decline of 1.5% on average compared to the best performance (the default), and negligible overhead in execution time and power consumption. We believe HiCAP presents an intuitively natural way to model and exploit the dynamic behavior of mobile games, and is easily portable across newer mobile platforms. The work presented here is our first step, with ongoing and future work addressing: 1) proposing a scientific methodology for dynamic behaviors such as classification and regression machine learning algorithms, and 2) integration of dynamic thermal management in addition to cooperative CPU/GPU power management techniques. Finally, while our HiCAP methodology was targeted mainly for mobile games, we believe it can also be applicable for various other classes of CPU-GPU integrated graphics applications.

Chapter 5

A Machine Learning Enhanced Integrated Governor

5.1 Introduction

Mobile games are an increasingly important application workload for mobile devices in terms of increasing number of game applications and dynamism of gaming workloads. The recent trend towards Heterogeneous MultiProcessor Systems-on-Chip (HMPSoC) architectures (e.g., ARM's big.LITTLE with integrated GPU) attempt to meet the performance needs of mobile devices, and rely on software governors for dynamic power management in the face of high performance. Besides separate governors for contemporary commercial CPU and GPU DVFS power management, some recent research efforts have proposed integrated CPU-GPU DVFS policies [80] [51] for a small set of mobile games, assuming fairly static gaming workloads. However, gaming applications exhibit inherent dynamism in their workloads, and recent research on software governors typically use classical statistical methods (e.g., simple linear regression models [80] with a small amount of specific training data), resulting

in high prediction errors for unseen workloads. These classical linear regression based approaches are not effective for capturing the non-linear dynamism of gaming applications, since they impose a linear relationship on the data [95]. In order to overcome the limitations of classical statistical models (e.g., assumption of linear relationship or considering a large number of variables), some recent approaches for GPGPUs [99] and High-Performance Computing (HPC) [27] have developed performance prediction models using machine learning techniques. But – to the best of our knowledge – machine learning enhanced approaches have not been investigated for CPU-GPU integrated governors managing gaming workloads on mobile heterogeneous MPSoCs.

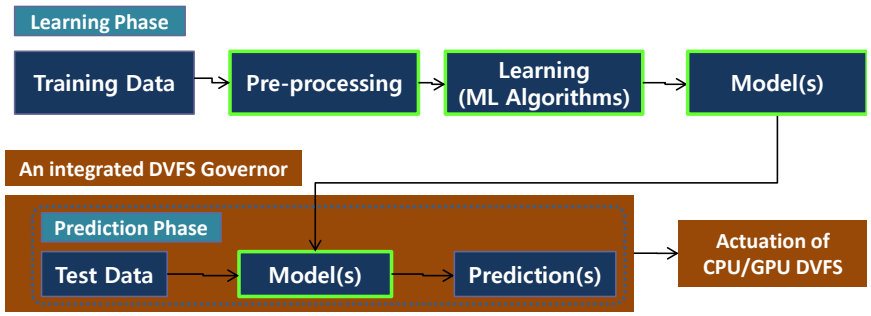


Figure 5.1: Machine Learning Approach for our System

To address these issues we propose ML-Gov: a machine learning enhanced integrated CPU-GPU DVFS governor (Figure 5.1) that proceeds in two phases: 1) in the learning phase, we build tree-based piecewise linear models with high accuracy and a simple cost function structure using practical offline machine learning techniques, and 2) we deploy these models into an integrated DVFS governor that achieves online runtime estimation using the models.

This work makes the following specific contributions:

- We propose a practical machine learning enhanced tree-based piecewise regression model building methodology for diverse and dynamic gaming workloads on heterogeneous mobile platforms
- We present an integrated CPU-GPU DVFS governor that applies piecewise policies

with analyses of the models.

- We present experimental results on a set of 20 mobile games with various characteristics, showing significant energy savings of over 10% in Energy-per-Frame (EpF) with a surprisingly concomitant 3% improvement in FPS performance, compared to a state-of-the-art governor.

The rest of the work is organized as follows: Chapter 5.2 gives motivation and related work. Chapter 5.3 distinguishes our methodology using the learning and prediction phases. Chapter 5.4 shows and analyzes our results. Finally Chapter 5.5 concludes with a summary and future work.

5.2 Motivation and Related Work

Unlike general machine learning based approaches, embedded systems pose two crucial challenges for machine learning: 1) model building considering system specific characteristics or constraints such as heterogeneous architectures (i.e., not just higher accuracy), and 2) a simple structure of cost functions for model integration and evaluation of the integrated system, within a resource-constrained platform.

With regard to the first challenge, a specific characteristic of our integrated governor is that our prediction models should be integrated into a CPU-GPU governor, and the models must estimate the appropriate CPU and GPU frequencies for each state while achieving the system goal of maximizing energy savings with minimal performance degradation. The second challenge requires that the cost functions of the models should be easily integrated into the governor algorithm, with negligible computational overhead within the time interval epochs of the CPU or GPU governors.

While there are many powerful machine learning techniques such as instanced-based learn-

ing (e.g., k-Nearest Neighbor) or neural networks that provide high accuracy, they also suffer from high opacity (not revealing anything about the structure of cost functions). On the other hand, the class of regression models provides simple cost functions but low accuracy.

To solve the challenge of predicting real values for non-linear type datasets, we deploy a tree-based piecewise linear model (i.e., model trees [95]), which combines a conventional decision tree with the possibility of linear regression functions at the leaves, that provides high accuracy with a simple cost function structure that allows for ease of integration into the governors. We note that the integrated governor using these prediction models may sometimes lead to unexpected results such as sudden performance drops (e.g., due to over-/under-fitting of the models, unseen dynamic workloads, or heuristic thresholds in a governor algorithm).

Therefore we found it important to deploy a simple and analyzable cost function structure that allows us to investigate and resolve the problems in the integrated system. Note that our approach is a combination of a decision tree with regression models at leaf nodes; the representation of our models is perspicuous because the decision structure is simple due to a small number (3-5) of leaf nodes in a tree and the regression functions do not involve many attributes due to feature selection.

5.2.1 Motivation

Now consider Figure 5.2, where we compare model accuracy (prediction errors) and complexity of model cost functions among the machine learning algorithms (Table 5.1) to motivate need of tree-based piecewise regression models. (For the motivating example in Figure 5.2, we use two datasets, which are collected from a real platform using a training set of 20 diverse games; and the characteristics of the dataset and platform configurations will be described in detail in Chapter 5.3.1 and 5.4.1).

The machine learning algorithms described in Table 5.1 are already built in a data-mining

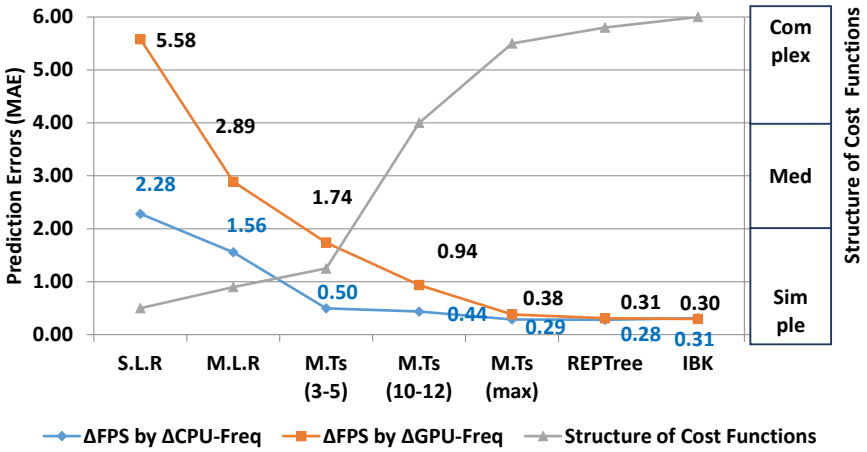


Figure 5.2: Comparison of Prediction Errors and Structure of Cost Functions among the Machine Learning Algorithms (Motivating Example)

tool called *Weka* [41]. ΔFPS *by* Δ*CPU-Freq* (Δ*GPU-Freq*) is FPS sensitivity to CPU (GPU) frequency change after feature selection. As a metric for prediction errors (the left y axis), we use Mean Absolute Error (MAE), which is a good indicator of average model performance [97]. The structure of cost functions (the right y axis) is compared relatively using simple, medium and complex levels among the algorithms. In other words, while an instance-based learning does not reveal anything about the structure of the function (highest complexity) because of the non-parametric property, a simple linear regression provides the simplest structure. For the model trees, if the decision tree is simple (i.e., a small number of leaf nodes) and the regression functions do not normally involve many variables after a feature selection, the representation is simple and analyzable. However, as the number of leaf nodes in a tree increases, the complexity will also increase as shown in the middle of Figure 5.2.

Table 5.1: Compared M.L Algorithms

Algorithms in Weka [41]	Descriptions
S.L.R	Simple Linear Regression
M.L.R	Multivariate Linear Regression
M5P [86] [95]	Model Trees (M.Ts)
REPTree	Decision (Classification/Regression) Trees
IBk [4]	k-Nearest Neighbor (k-NN)

For instance, while the average MAE of S.L.R and M.L.R for ΔFPS *by* Δ*CPU-Freq* (i.e.,

FPS sensitivity to CPU frequency change) is 2.28 and 1.56 respectively, that of M.T using 3-5 leaf nodes is 0.5 (i.e., the MAE of the M.T was reduced by 78% and 68% compared to S.L.R and M.L.R respectively). From the perspective of cost function complexity, M.Ts (3-5) after a feature selection have almost similar complexity with the multivariate linear models. This comparison clearly shows the need for the model trees in terms of simple and analyzable structure of cost functions with high accuracy.

5.2.2 Related Work

With the emergence of high performance integrated mobile GPUs, several research efforts have proposed integrated CPU-GPU DVFS governors: Pathania *et al.*'s [82] integrated CPU-GPU DVFS algorithm didn't consider quantitative evaluation for energy savings (e.g., per-frame energy or FPS per watt); their next effort [80] further developed power-performance models to predict the impact of DVFS on mobile gaming workloads, but used simple linear regression models using a small amount of specific data resulting in high prediction errors for various unseen workloads. Kadjo *et al.* [51] used a queuing model to capture CPU-GPU-Display interactions across a narrow range of games exhibiting limited diversity in CPU-GPU workloads; Park *et al.* [77] proposed a coordinated CPU-GPU maximum frequency capping technique and applied it to a diverse range of mobile graphics workloads, but assumed static characterization of each application; their next effort [78] further developed a hierarchical FSM-based dynamic behavior modeling strategy for mobile gaming considering QoS and CPU/GPU workload dynamism, but used an adaptive frequency-capping technique on top of the default CPU and GPU governors instead of proposing a prediction model based integrated frequency scaling technique.

Recently, CPU/GPU performance or power estimation models that use machine learning techniques are emerging in order to overcome these challenges: model building from training data at numerous different hardware configurations for diverse and dynamic appli-

cations (workloads) on high-performance cluster (HPC) or general purpose GPU (GPGPU) platforms. Dwyer *et al.* [27] proposed a method for estimating performance degradation on multicore processors and applied into HPC workloads; Wu *et al.* [99] presented a GPGPU power and performance estimation model that uses machine learning techniques on measurements from real hardware performance counters. However, both do not consider a simple cost function structure for easy integration of the models into a system and also do not analyze the effects of the models during runtime prediction; instead these efforts focus purely on high accuracy.

Gupta *et al.* [40] introduced a need for online performance models that can adapt to varying workloads since the impact of the GPU frequency on performance varies rapidly over time and presented a light-weight adaptive runtime performance model that predicts the frame processing time; they did not present an integrated CPU-GPU governor but only introduced potential impacts for GPU dynamic power management using the model. Chuan *et al.* [20] proposed an adaptive on-line CPU-GPU governor for games on mobile devices to minimize energy consumption. However, their work was applied to a set of only three games exhibiting a narrow range of CPU-GPU workloads; and they did not show applicability across a wide range of games exhibiting diverse CPU-GPU workloads in spite of significantly different results from different types of graphics workloads.

To the best of our knowledge, our work – unlike previous efforts focused purely on performance without regard to generality for mobile systems – is the first to introduce a practical machine-learning enhanced piecewise linear model building methodology that achieves high accuracy while using a simple cost function, allowing for ease of integration into CPU-GPU integrated governors for mobile platforms. We therefore aim to achieve a target FPS with minimal total power consumption using an integrated CPU-GPU DVFS. Furthermore, we present experimental results for an integrated CPU-GPU governor applied on mobile gaming workloads using a test set of 20 mobile games exhibiting diverse characteristics executing on a mobile HMPSoC platform.

5.3 ML-Gov Methodology

Our ML-Gov methodology has two phases: a learning phase and a prediction phase as shown in Figure 5.3.

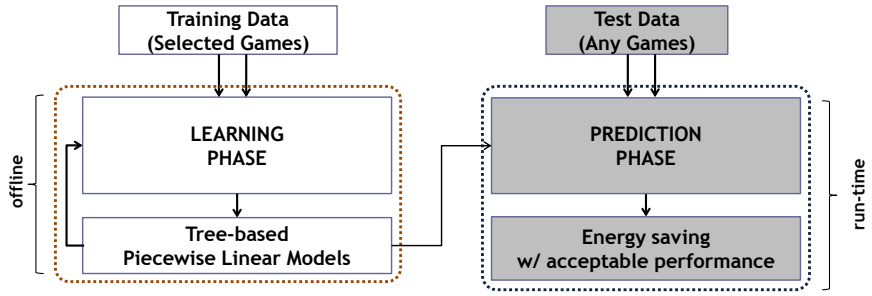


Figure 5.3: ML-Gov Overview

In the learning phase, we build tree-based piecewise linear regression models as well as comparable statistical models using offline machine learning techniques built in a data mining tool called Weka [41]. Then in the prediction phase, ML-Gov uses the built models at runtime to estimate appropriate CPU and GPU frequencies (as much as possible maximizing energy savings with minimal FPS degradation).

The main goal of this work is to present a new practical model-building approach using offline machine learning techniques evaluating model accuracy and structure of cost functions in the learning phase; and then we evaluate the real effects of the prediction models in terms of energy saving and performance (FPS) in the prediction phase. Therefore, for an integrated CPU-GPU governor framework, we deploy a simple but already qualified integrated governor framework using a hierarchical-FSM based representation based on thorough observations of state-of-the-art power management techniques [80] [78].

5.3.1 Learning Phase

As shown in Figure 5.4, the learning phase is composed of three steps: 1) collection of training data, 2) attribute selection and 3) model training. The main objective of this phase

is to build prediction models with simple complexity of cost functions and high accuracy to integrate the models easily into an integrated CPU-GPU DVFS governor.

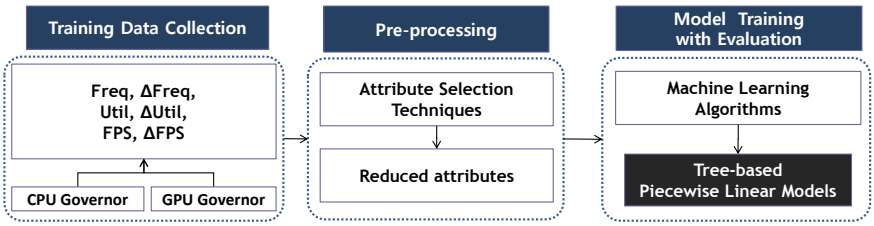
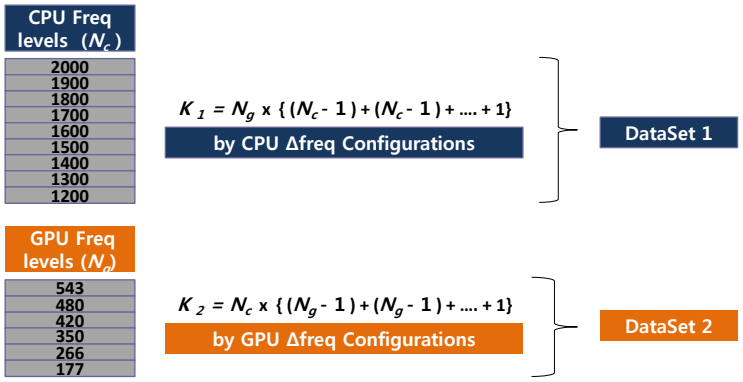


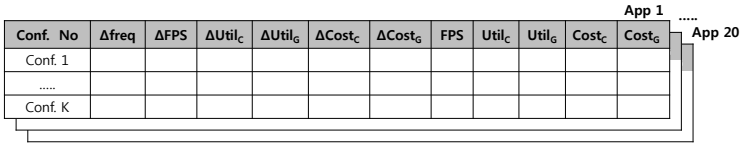
Figure 5.4: Learning Phase

5.3.1.1 Collection of Training Data

For training data, we use a training set of 20 mobile games that has various characteristics in terms of CPU- and GPU-workloads using a workload metric (*Cost*) that is a product of the utilization and frequency [16] [82] [77] [78]; and we collect data at numerous different CPU and GPU frequency configurations using the diverse applications (workloads) on a mobile HMPSoC platform.



(a) Frequency Configurations



(b) DataSet

Figure 5.5: Data Collection Methodology

As shown in Figure 5.5.(a), we need two datasets because our system design has two actu-

ators, CPU and GPU DVFS. Therefore, first we measure a raw dataset of FPS, CPU-GPU frequencies and utilizations across a range of frequency configurations by sweeping the CPU frequency across the set of frequencies supported by the target system (9 frequency levels in CPU) at each GPU frequency of 6 frequency levels in GPU; and we measure another raw dataset by sweeping the GPU frequency with the same methodology. And then we collect two datasets using the values of two different CPU/GPU frequency configurations from the raw datasets assuming one is a current and the other is a new frequency. Specifically, we choose 11 crucial variables (the first row in Figure 5.5.(b)) based on comprehensive observations of related work [82] [80] [51] [77] [78] for mobile gaming characteristics to CPU/GPU frequency: 1) Relationship between FPS and CPU/GPU frequency. 2) Relationship between a component's utilization and its frequency. 3) The impact of cross-component frequency variations on utilizations. 4) The impacts of CPU-GPU *Cost* functions. 5) The impacts of current FPS, CPU-GPU utilizations and *Cost* functions.

As a result, in the training phase, we collect two datasets of 7020 = 20 x (36x6 + 15x9) instances with 351 different frequency configurations (216 for CPU and 135 for GPU frequency effects).

5.3.1.2 Attribute Selection

Before training the models, we reduce the number of attributes in the datasets for two reasons: 1) to remove the attributes that are redundant or unrelated to an output, and 2) to reduce the complexity of cost functions and computation overhead, while maintaining a good prediction accuracy. Choosing a specific technique for attribute selection can be dependent on the data and application area. For our model (i.e., FPS and Utilization prediction by CPU and GPU frequency changes), the most important factor to consider is the correlation between one response attribute and the other attributes. We tested several attribute selection techniques by constructing models using the two datasets (Figure 5.5) and comparing their

accuracy; the technique that achieved the lowest error rate was *correlation based feature subset attribute selection (CfsSubset)* [42] which sorts the attributes by their correlation to a class attribute (response variable) and to the other attributes (predictor variables) in the dataset. Note that the number of selected predictor variables may be different from each response variable by the property of *CfsSubset* algorithm; Table 5.2 shows the results of attribute selection.

Table 5.2: Selected Variables after Attribute Selection

Response Variable	Selected Predictor Variables
ΔQ by ΔF_C	ΔF_C, Q, U_C and C_G
ΔU_C by ΔF_C	ΔF_C, Q, U_G, C_C and C_G
ΔU_G by ΔQ_{F_C}	ΔQ
ΔQ by ΔF_G	F_G (ΔF_G), Q, U_C and U_G
ΔU_G by ΔF_G	ΔF_G and C_C
ΔU_C by ΔQ_{F_G}	ΔQ and C_G

5.3.1.3 Model Training

Next, we build and evaluate models for the qualified algorithms introduced in Table 5.1. Here we detail the analyses and choices made for model evaluation and training.

First, to build models for the six responsive variables in Table 5.2, we mainly train the first three algorithms among the algorithms described in the motivating example: S.L.R, M.L.R and M.Ts (3-5). We note that instance-based learning does not reveal anything about the structure of cost functions due to the property of a non-parametric method; and that regression trees approximating a non-linear function by discretizing hundreds of leaves also are not adequate for our integrated governor. Furthermore, M.Ts (max) are evaluated for comparing model accuracy (but not for using the built models), since it is almost impossible to integrate dozens of model trees into the CPU-GPU governor and analyze the effects of the models.

We then build models of M.Ts (3-5) by changing *minNumInstances* (the minimum number of instances allowed at a leaf node) in the M5P algorithm. We empirically choose a model tree that has the smallest number among 3-5 leaf nodes, because it has a simple (analyzable)

cost function structure while its prediction errors are significantly lower than those of M.L.R or the model trees having only 2 leaf nodes.

Table 5.3: Prediction Errors for Model Evaluation

Response Var.	S.L.R	M.L.R	C-M.T	F-M.T	Chosen
ΔQ by ΔF_C	3.79%	2.58%	0.83% (3)	0.48%	M.T 1
ΔU_C by ΔF_C	15.55%	12.73%	11.15% (5)	6.33%	M.T 2
ΔU_G by ΔQ_{F_C}	4.49%	4.49%	4.44% (5)	4.46%	S.L.R 1
ΔQ by ΔF_G	9.96%	5.15%	3.10% (4)	0.68%	M.T 3
ΔU_G by ΔF_G	19.82%	15.24%	14.04% (4)	8.42%	M.T 4
ΔU_C by ΔQ_{F_G}	22.78%	22.71%	22.55% (5)	16.59%	S.L.R 2

Table 5.3 summarizes the results of our model evaluation. The last column of this table shows our final selection, comprising four M.Ts (1-4) and two S.L.Rs (1-2) for the six responsive variables. These selections were made based on their lower prediction errors and simple cost function structures: Model prediction errors for ΔU_G by ΔQ_{F_C} and ΔU_C by ΔQ_{F_G} are almost similar for all algorithms so that we use the simplest S.L.R models However, for the other response variables, we use the M.Ts (1-4) because of high accuracy and analyzable cost functions.

We illustrate the structure of the Model Trees using M.T 1 as an example. Model Tree M.T 1 is a tree-based piecewise linear regression model to predict ΔQ by ΔF_C. Lines 1-4

Model Tree M.T 1: ΔQ **by** ΔF_C

1: $U_C \leq 82$: LM1 (2774/16.283%)
2: $U_C > 82$:
3: — $Q \leq 54$: LM2 (1160/57.601%)
4: — $Q > 54$: LM3 (386/20.231%)

▷ **Smoothed Mode**
5: LM1: ΔQ = - 0.0007 * Q + 0.0003 * U_C - 0.001 * C_G + 0.0603
6: LM2: ΔQ = 0.016 * ΔF_C - 0.0042 * Q + 0.0005 * U_C + 0.4079
7: LM3: ΔQ = 0.0004 * ΔF_C - 0.0098 * Q + 0.0005 * U_C + 0.8146

▷ **Un-smoothed Mode**
8: LM1: ΔQ = -0.001 * C_G + 0.0407
(we set -0.001=α_1^{MT1}, 0.0407=α_2^{MT1})
9: LM2: ΔQ = 0.0162 * ΔF_C + 0.2358
(we set 0.0162=β_1^{MT1}, 0.2358=β_2^{MT1})
10: LM3: ΔQ = + 0.3811
(we set 0.3811=γ_1^{MT1})

reveal a tree structure with the thresholds of U_C and Q, and Lines 5-7 and 8-10 correspond to each regression model in each leaf node with the parameters in smoothed and un-smoothed

modes. In our model, we use un-smoothed mode instead of smoothed mode because the structure of the un-smoothed mode is simpler while the prediction errors between the two modes are exactly the same in our datasets.

Model Tree 2: ΔU_C **by** ΔF_C

1: $C_C \leq 61.152$
2: — $\Delta F_C \leq 250$: LM1 (1293/54.063%)
3: — $\Delta F_C > 250$:
4: — — $C_C \leq 54.416$: LM2 (1549/52.608%)
5: — — $C_C > 54.416$:
6: — — — $C_G \leq 29.714$: LM3 (162/9.39%)
7: — — — $C_G > 29.714$: LM4 (243/198.57%)
8: $C_C > 61.152$: LM5 (1073/22.061%)

9: LM1: ΔU_C = -0.018 * ΔFc - 0.0585 * Q - 0.0294 * C_C + 4.1886
(we set -0.018=α_1^{MT2}, - 0.0585=α_2^{MT2}
- 0.0294=α_3^{MT2}, 4.1886=α_4^{MT2})
10: LM2: ΔU_C = -0.0186 * ΔFc - 0.1521 * C_C - 0.0415 * C_G + 7.4911
(we set -0.0186=β_1^{MT2}, - 0.1521=β_2^{MT2}
- 0.0415=β_3^{MT2}, 7.4911=β_4^{MT2})
11: LM3: ΔU_C = -0.1934
(we set -0.1934=γ_1^{MT2})
12: LM4: ΔU_C = -14.7754
(we set -14.7754=θ_1^{MT2})
13: LM5: ΔU_C = -0.0078 * U_G - 0.0534 * C_G + 1.5904
(we set -0.0078=λ_1^{MT2}, - 0.0534=λ_2^{MT2}, 1.5904=λ_3^{MT2})

The Model Tree 2 is a tree-based piecewise linear regression model to predict ΔU_C by ΔF_C. Line 1-8 reveals a tree structure with 5 leaf nodes by the thresholds of C_C, C_G and ΔF_C, (if ΔF_C or ΔF_G is included in the thresholds, we estimate it using another multivariate linear regression model); and Line 9-13 corresponds to each regression model in each leaf node with the parameters in the un-smoothed mode.

Model Tree 3: ΔQ **by** ΔF_G

1: $U_G \leq 96.86$:
2: — $U_G \leq 39.608$: LM1 (171/3.352%)
3: — $U_G > 39.608$:
4: — — $U_C \leq 96.032$: LM2 (395/12.61%)
5: — — $U_C > 96.032$: LM3 (212/12.486%)
6: $U_G > 96.86$: LM4 (933/60.453%)

7: LM1: ΔQ = + 0.0282
(we set 0.0282=α_1^{MT3})
8: LM2: ΔQ = + 0.7848
(we set 0.7848=β_1^{MT3})
9: LM3: ΔQ = - 0.4865
(we set -0.4865=γ_1^{MT3})
10: LM4: ΔQ = 0.031 * ΔF_G - 0.55 * Q + 29.9607
(we set 0.031=θ_1^{MT3}, - 0.55=θ_2^{MT3}, 29.9607=θ_3^{MT3})

The Model Tree 3 is a tree-based piecewise linear regression model to predict ΔQ by

ΔF_G.

The Model Tree 4 is a tree-based piecewise linear regression model to predict ΔU_G by ΔF_G.

Model Tree 4: ΔU_G **by** ΔF_G
1: $\Delta F_G \leq 142$:
2: — $\Delta F_G \leq 66.5$: LM1 (360/13.196%)
3: — $\Delta F_G > 66.5$:
4: — — $U_G \leq 99.148$: LM2 (704/42.046%)
5: — — $U_G > 99.148$: LM3 (196/25.485%)
6: $\Delta F_G > 142$: LM4 (1440/88.951%)
7: LM1: ΔU_G = - 3.8548 (we set - 3.8548=α_1^{MT4})
8: LM2: ΔU_G = - 10.8804 (we set - 10.8804=β_1^{MT4})
9: LM3: ΔU_G = - 2.429 (we set - 2.429=γ_1^{MT4})
10: LM4: ΔU_G = -0.0896 * ΔF_G - 1.7533 (we set -0.0896=θ_1^{MT4}, - 1.7533=θ_2^{MT4})

And, the S.L.Rs for the two responsive variables (ΔU_G by ΔQ_{F_C} and ΔU_C by ΔQ_{F_G}) are as follows.

S.L.R 1: ΔU_G **by** ΔQ_{F_C}	
ΔU_G = 0.8704 * ΔQ	(we set 0.8704 = α_1^{SLR1})
S.L.R 2: ΔU_C **by** ΔQ_{F_G}	
ΔU_C = 1.4575 * ΔQ	(we set 1.4575 = α_1^{SLR2})

Model Equations: We derive model euquations from the four model trees and the two S.L.Rs. We denote the current CPU-GPU frequency combination with (F_C, F_G), the utilization values with $U_C^{(F_C,F_G)}$, $U_G^{(F_C,F_G)}$, the FPS at the frequency combination with $Q^{(F_C,F_G)}$ and the current CPU-GPU *Cost* with C_C, C_G.

To estimate the FPS at a higher CPU (GPU) frequency level F'_C (F'_G), the relationship between FPS and CPU (GPU) frequency can be derived by using Model Tree 1 (Model Tree 3) as follows (ΔF_C is same with F'_C - F_C, vice versa for ΔF_G).

$$Q^{(F'_C,F_G)} - Q^{(F_C,F_G)} = \begin{cases} \alpha_1^{MT1} C_G + \alpha_2^{MT1} & \text{if } LM1. \\ \beta_1^{MT1} \Delta F_C + \beta_2^{MT1} & \text{if } LM2. \\ \gamma_1^{MT1} & \text{if } LM3. \end{cases} \tag{5.1}$$

$$Q^{(F_C,F'_G)} - Q^{(F_C,F_G)} = \begin{cases} \alpha_1^{MT3} & \text{if } LM1. \\ \beta_1^{MT3} & \text{if } LM2. \\ \gamma_1^{MT3} & \text{if } LM3. \\ \theta_1^{MT3}\Delta F_G + \theta_2^{MT3} Q + \theta_3^{MT3} & \\ & \text{if } LM4. \end{cases} \tag{5.2}$$

To estimate the utilization of a component at a higher CPU (GPU) frequency level F'_C (F'_G), the relationship between a component's utilization and its frequency can be derived as follows, by using the Model Tree 2 (Model Tree 4).

$$U_C^{(F'_C,F_G)} - U_C^{(F_C,F_G)} = \begin{cases} \alpha_1^{MT2}\Delta F_C + \alpha_2^{MT2} Q + \alpha_3^{MT2} C_C + \alpha_4^{MT2} & \\ & \text{if } LM1. \\ \beta_1^{MT2}\Delta F_C + \beta_2^{MT2} C_C + \beta_3^{MT2} C_G + \beta_4^{MT2} & \\ & \text{if } LM2. \\ \gamma_1^{MT2} & \text{if } LM3. \\ \theta_1^{MT2} & \text{if } LM4. \\ \lambda_1^{MT2} U_G + \lambda_2^{MT2} C_G + \lambda_3^{MT2} & \text{if } LM5. \end{cases} \tag{5.3}$$

$$U_G^{(F_C,F'_G)} - U_G^{(F_C,F_G)} = \begin{cases} \alpha_1^{MT4} & \text{if } LM1. \\ \beta_1^{MT4} & \text{if } LM2. \\ \gamma_1^{MT4} & \text{if } LM3. \\ \theta_1^{MT4}\Delta F_G + \theta_2 & \text{if } LM4. \end{cases} \tag{5.4}$$

Finally, we estimate the impact of cross-component frequency variations on utilizations, which occurs as long as there is parallel increase in FPS [80]. Therefore, the corresponding equations can be derived as follows, by using the S.L.R2 (ΔU_C by ΔQ_{F_G}) and S.L.R1 (ΔU_G

by ΔQ_{F_C}).

$$U_C^{(F_C,F'_G)} - U_C^{(F_C,F_G)} = \begin{cases} \alpha_1^{S.L.R1}(Q^{(F_C,F'_G)} - Q^{(F_C,F_G)}) \end{cases} \tag{5.5}$$

$$U_G^{(F'_C,F_G)} - U_G^{(F_C,F_G)} = \begin{cases} \alpha_1^{S.L.R1}(Q^{(F'_C,F_G)} - Q^{(F_C,F_G)}) \end{cases} \tag{5.6}$$

5.3.2 Prediction Phase

As shown in Figure 5.6, the prediction phase comprises two steps: 1) merging the models into an integrated CPU-GPU DVFS governor framework, and 2) setting CPU and GPU frequencies by exeuting these predictors at runtime.

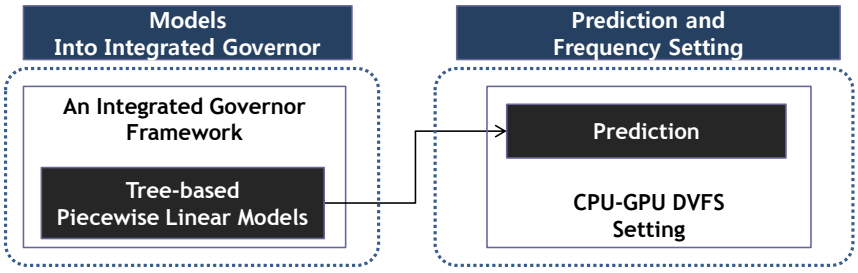


Figure 5.6: Prediction Phase

5.3.2.1 An Integrated Governor Framework

The objective of our manager is to achieve a target FPS as much as possible with minimal total power consumption using an integrated CPU-GPU DVFS. We build on the non-trivial observations gleaned from the related work [80] [77] [78]: it is more power/energy efficient to run at higher utilization and lower frequency than lower utilization and higher frequency if a current FPS achieves a target FPS (or a current FPS is already quantitatively competitive performance). However, for a target-FPS based manager, the challenge is how to get the information of a target maximum FPS or reference values such as maximum CPU and GPU

utilizations for quantitative comparison with other governors. To solve this issue (similar to the previous work [80]), we take three samples (one second duration for each) to obtain the game specific reference constants ($\hat{Q}$, $\hat{U}c$, $\hat{U}g$) when a new scene starts with the assumption that start of a scene can be detected by changes in rendered textures [25] or CPU-GPU utilization patterns [82]. This kind of sampling method enables us to avoid prior comprehensive offline profiling [80], especially to compare with the default separate CPU and GPU governors (i.e., performance-driven policy without CPU-GPU cooperation).

Figure 5.7 illustrates our power management algorithm using the Hierarchical FSM-based representation [78], since it provides a natural and intuitive design abstraction. Our algo-

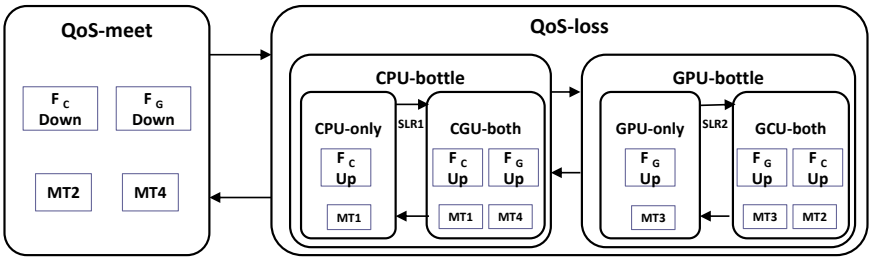


Figure 5.7: HFSM-based Power Management Algorithm

rithm detects two possible super-states at current frequency combination (F_C, F_G): if Q is within $\hat{Q}$, *QoS-meet* or if Q is less than $\hat{Q}$, *QoS-loss*. For *QoS-meet* super-state, CPU and GPU frequencies will be scaled down to the estimated frequencies using the tree-based linear models to achieve the maximum utilizations (using M.T 2 and 4). For *QoS-loss* super-state, there are two sub-states: if CPU is the bottleneck, *CPU-bottle* or if GPU is the bottleneck, *GPU-bottle*. *CPU-bottle* sub-state has two leaf states: *CPU-only* and *CGU-both*. For *CPU-only* leaf state, only CPU frequency will be scaled up to the estimated frequency to achieve the maximum target-FPS (using M.T 1); for *CGU-both* leaf state, GPU frequency will be scaled up to the estimated frequency (using M.T 4) in addition to the CPU frequency scale up. (Vice versa for *GPU-bottle* sub-state: For *GPU-only* leaf state, only GPU frequency will be scaled up to the estimated frequency to achieve the maximum target-FPS (using M.T 3); for *GCU-both* leaf state, CPU frequency will be scaled up to the estimated frequency (using

M.T 2) in addition to the GPU frequency scale up.)

5.3.2.2 CPU/GPU Frequency Setting with Prediction

Performance Demand State: For the *QoS-loss* super-state, the sub-state is *CPU-bottle* or *GPU-bottle*, which reuire an increase in the frequency of the bottlenecked component for FPS improvement. Let the needed frequency combination be (F'_C, F'_G), where $F'_C \geq F_C$ and $F'_G \geq F_G$. For the *CPU-only* leaf-state, we choose F'_C using Equation (7), derived from Equation (1): For LM 2, the equation predicts a next CPU frequency achieving the maximum target-FPS. However, if ΔF_C is not critical variable to the FPS sensitivity, we apply an adaptive heuristic policy (i.e., one step higher frequency) for LM1 and LM3. According to our observations, when we apply the same CPU frequency instead of one step higher frequency, some intensive applications result in significant performance degradation.

$$F'_C = \begin{cases} \frac{(\hat{Q} - Q^{(F_C, F_G)}) - \beta_2^{MT1}}{\beta_1^{MT1}} + F_C & \text{if } LM = 2. \\ F_C{+}{+} & \text{otherwise.} \end{cases} \tag{7}$$

This increase in FPS may force the cross-component (GPU) to do more work increasing its utilization; and even after increasing CPU frequency, it may fail to achieve $\hat{Q}$ because of an intermediate GPU bottleneck. The estimated U_G at $\hat{Q}$ would be given by the Equation (8), based on Equation (5).

$$U_G^{(F'_C, F_G)} = U_G^{(F_C, F_G)} + \alpha_1^{SLR1}(\hat{Q} - Q^{(F_C, F_G)}) \tag{8}$$

If $U_G^{(F'_C, F_G)}$ is greater than $\hat{U}_G$, GPU will also become a bottleneck and the state will change to *CGU-both* leaf-state; and the GPU frequency should be increased to F'_G given by Equation

(9), derived from Equation (4) (the Model Tree 4).

$$F'_G = \begin{cases} \frac{(U_G^{(F_C,F_G)} - \hat{U}_G) - \theta_2^{MT4}}{\theta_1^{MT4}} + F_G & \text{if } LM = 4. \\ F_G{+}{+} & \text{otherwise.} \end{cases} \tag{9}$$

For *GPU-bottle* sub-state, first we estimate F'_G using Equation (2) (the Model Tree 3) for *GPU-only*. Second, we can detect the condition to *GCU-both* leaf-state using Equation (6) in S.L.R 2. And then, F'_C can be estimated using Equation (3) (the Model Tree 2) for *GCU-both* leaf-state.

Power Saving State: For the *QoS-meet* super-state, already the maximum target-FPS is achieved with over-provisioned CPU and GPU frequencies wasting power consumption. Therefore, we can save power without quality loss by reducing CPU and GPU frequencies to F''_C and F''_G achieving the maximum CPU and GPU utilizations: Equation (10) is derived from Equation (3) (the Model Tree 2), and Equation (11) is derived from Equation (4) (the Model Tree 4).

$$F''_C = \begin{cases} \frac{(\hat{U}_C - U_C^{(F_C,F_G)}) - (\alpha_2^{MT2} * Q + \alpha_3^{MT2} * C_C + \alpha_4^{MT2})}{\alpha_1^{MT2}} + F_C & \\ & \text{if } LM = 1. \\ \frac{(\hat{U}_C - U_C^{(F_C,F_G)}) - (\alpha_2^{MT2} * C_C + \alpha_3^{MT2} * C_G + \alpha_4^{MT2})}{\alpha_1^{MT2}} + F_C & \\ & \text{if } LM = 2. \\ F_G{+}{+} & \text{otherwise.} \end{cases} \tag{10}$$

$$F''_G = \begin{cases} \frac{(\hat{U}_G - U_G^{(F_C,F_G)}) - \theta_2^{MT4}}{\theta_1^{MT4}} + F_G & \text{if } LM = 4. \\ F_G{+}{+} & \text{otherwise.} \end{cases} \tag{11}$$

5.4 Experimental Results

5.4.1 Experimental Setup

We evaluated our ML-Gov manager on the ODROID-XU3 development board installed with Android 4.4.2 and Linux 3.10.9; Table 5.4 summarizes our platform configuration. The platform is equipped with four TI INA231 power sensors measuring the power consumption of big CPU cluster (CPU-bc), little CPU cluster (CPU-lc), GPU and memory respectively. The CPU supports cluster-based DVFS at nine frequency levels (from 1.2Ghz to 2.0Ghz) in CPU-bc and at seven frequency levels (from 1.0Ghz to 1.6Ghz) in CPU-lc, and GPU supports six frequency levels (from 177Mhz to 543Mhz).

Table 5.4: Platform Configuration

Feature	Description
Device	ODROID-XU3
SoC	Samsung Exynos5422
CPU	Cortex-A15 2.0Ghz and Cortex-A7 Octa-core CPUs
GPU	Mali-T628 MP6, 543Mhz
System RAM	2Gbyte LPDDR3 RAM at 933MHz
Mem. Bandwidth	up to 14.9GB/s
OS(Platform)	Android 4.4.2
Linux Kernel	3.10.9

Benchmark Set: We use a training set of 20 games and a test set of 20 games including several micro-benchmarks, covering different types of graphics workloads.

Figures 5.8 and 5.9 summarize the 20 training games and the 20 test set games used in our experiments. Note that these games were selected specifically to exercise different combinations of CPU- and GPU- intensive workloads, and included not only a large number of real games, but also some custom micro-benchmarks in order to cover the entire space of different graphics workloads.

The CPU-GPU graphics workload varation and their relative intensity is quantified using a CPU-GPU cost metric that is a product of the utilization and frequency [16] [82] [77], as shown by the rows and columns of the tables in Figures 5.8 and 5.9.

Normalized Cost CPU \ GPU	0–20 (0)	20–40 (1)	40–60 (2)	60–80 (3)	80–90 (4)	90–100 (5)
0–20 (0)		Mb_m01				
20–40 (1)		Dhoom 3 AVP Evolution	Bike Rider, Godzilla Mb_m12	Mb_m13	Mb_m24	3dmark Extream
40–60 (2)		Mb_m21 Modern Comb	D - day	3dmark normal		
60–80 (3)	Call of Duty	Jet Ski 2013			Mb_m44	
80–90 (4)				Edge of Tomorrow		
90–100 (5)		Turbo FAST Mb_m52 Q3zombie-M4				20 Apps (TrainingSet)

Figure 5.8: The 20-App Training Set

Normalized Cost CPU \ GPU	0–20 (0)	20–40 (1)	40–60 (2)	60–80 (3)	80–90 (4)	90–100 (5)
0–20 (0)		Anomaly2 low	Dream Bike			Action Bike
20–40 (1)		Deerhunter14,	Citadel Herculous	Anomaly2 normal	Anomaly2 high	
40–60 (2)	Q3zombie-M1	300, Mb102	Mb111, Mb112			
60–80 (3)		Mb101 Mb103	Mb113		GPU Bench	
80–90 (4)			Real Driving	Robocop		
90–100 (5)		Q3zombie-M2				20 Apps (TestSet)

Figure 5.9: The 20-App Test Set

We note that the gaming applications located on the right-side of the diagonal line represent a GPU-dominant workload (since GPU cost is higher than CPU cost); similarly, games on the left of the diagonal line are CPU-dominant.

Furthermore, higher values of the cost ratio represent more intensive CPU or GPU workloads, with the highest (e.g., 90-100 (level 5)) representing the most CPU- or GPU- intensive gaming applications or benchmarks.

In this context, the 20 test set gaming applications analyzed in Figure 5.11 can be interpreted using the cost matrix model in Figure 5.9; we attempted to test games exhibiting a variety of CPU- and GPU- intensity, as summarized below:

- CPU-Domi + GPU-Mix applications such as mb111, mb112 and Robocop have higher

GPU cost values compared to CPU-intensive workloads such as mb101, mb103 and Q3zombie-M2.

- For Mem-Mix applications, the CPU-GPU cost matrix model does not include the memory cost values. Therefore, we additionally present the memory cost values of Mem-Mix applications such as mb103 (62%), mb113 (72%) and Robocop (77%), compared to non-memory intensive applications such as mb101 (37%) and mb111 (39%).

Automatic Measurement Tool: To automatically measure a large set of mobile games quantitatively, we developed an automatic measurement tool implemented using python modules, xml files, and Linux shell scripts. Using this tool, we captured the average values of FPS, total power (CPU-bc, CPU-lc and GPU) and energy per frame (EpF) for three runs (120 seconds for each run) of each benchmark, and averaged their measurements.

For comparison: We compare our M.L-gov manager against the default, a state-of-the-art governor [80], adaptive only and model-tree only policies as shown in Table 5.5.

Table 5.5: Governors for Comparison

Governor	Description
Default	Interactive CPU and proprietary GPU governor
PAT15	Simple Linear Regression-based Governor
ML-Gov	Our Tree-based piecewise Linear Regression
Naive-Adap.	Naive one step higher/lower Adaptive policy
Naive-M.T	Only Tree-based Linear Models w.o Adaptation

The default corresponds to the independent CPU and GPU governors in Linux (Interactive CPU governor and ARM's Mali Midgard GPU governor). The state-of-the-art governor [80] (represented as PAT15) proposed an integrated CPU-GPU DVFS strategy by developing predictive simple linear regression power-performance models. The naive adaptive governor is represented as Naive-Adap in the figures which does not use the built model-trees but adaptively scale up to one step higher (or down to one step lower) frequency for corresponding component according to each leaf state. The naive model-tree governor is represented as Naive-M.T, which does not use the heuristic adaptation (i.e., one step higher or lower) but

utilizes the built model-trees based on the assumption that offline learning models generalize all possible workloads that would be executed by the system. In other words, if ΔF_C or ΔF_G is not related to a response variable in the models, we set the next frequency same with the current frequency (i.e., $F'_C = F_C$ or $F'_G = F_G$) for any states without using the heuristic adaptation.

Overhead and Epoch of our Manager: Our power manager was implemented in the Linux kernel layer. The execution time of our manager per epoch is within 20us, which is totally negligible compared to the epoch period (200ms) in terms of performance (FPS degradation). Moreover, we did not observe any noticeable increase in average power consumption due to our power manager. And, the reason we use the epoch of 200ms (double of the GPU governor epoch) is that a longer epoch may affect performance degradation because of delayed frequency settings while it can provide more accurate FPS information (for below 60 FPS) per epoch.

5.4.2 Results and Analysis

Figure 5.10 summarizes the average results of the test set in FPS and EpF. Our ML-Gov manager improves energy per frame by 9.5% and 10.6% on average compared to the default and PAT15 respectively, with minimal FPS decline of 3.5% compared to the default and 2.7% better FPS than PAT15 on average. The two naive governors have 7.8% better EpF on average compared to our governor, but it results from the high FPS degradation of 7% (up to 32%) compared to the Default.

Figure 5.11 shows the results of each application and summary of CPU- and GPU-dominant applications. Especially, compared to PAT15, Figure 5.10 and 5.11 clearly show that tree-based piecewise linear regression model has significant impacts on energy savings with FPS improvement in the prediction phase, in addition to the improvements of the model

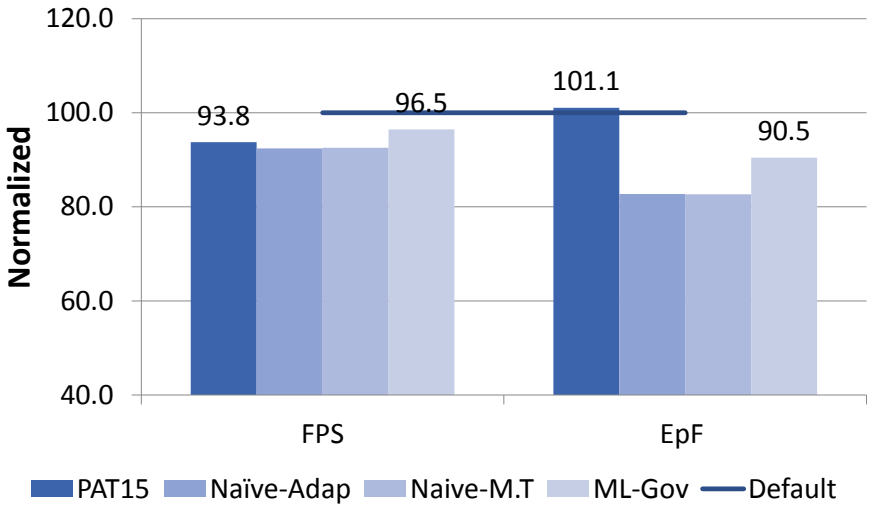
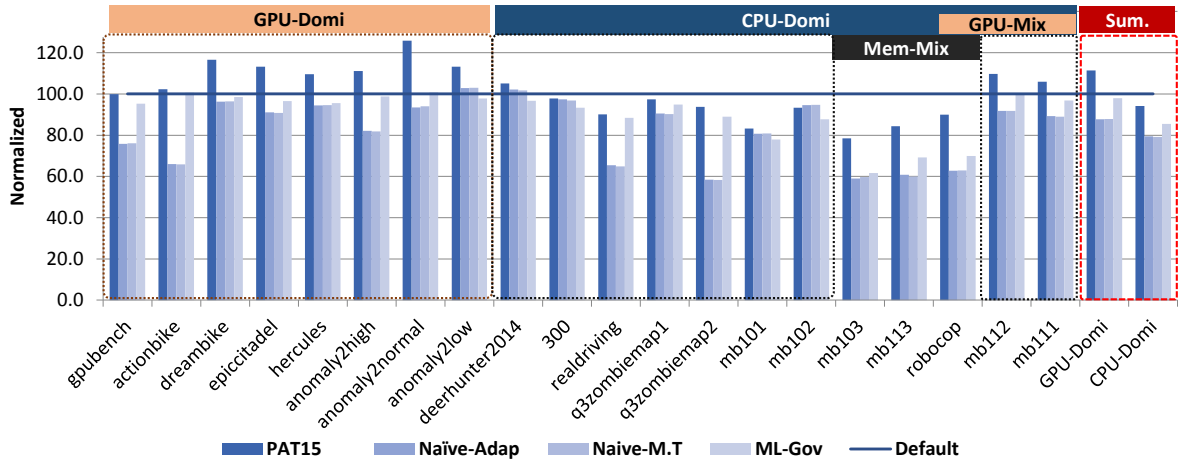


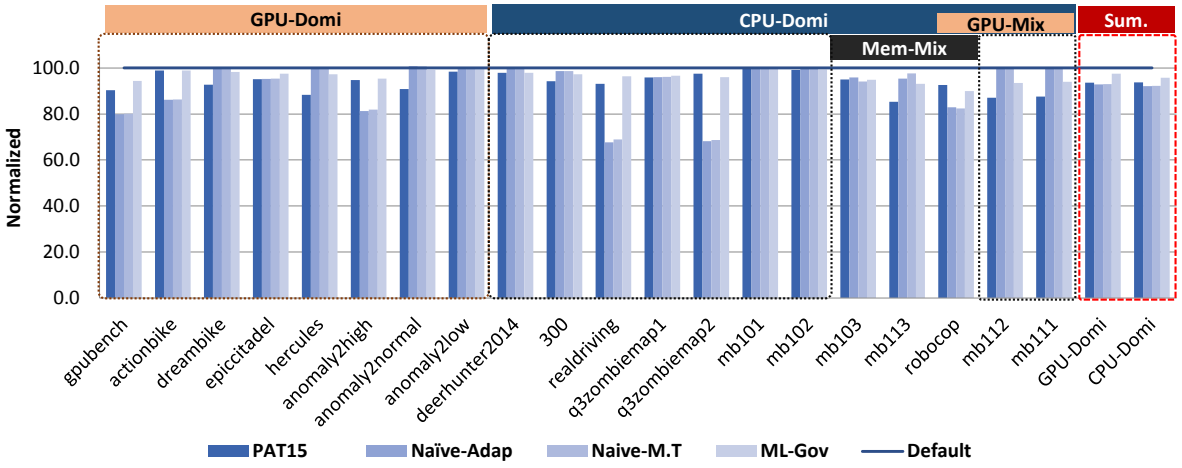
Figure 5.10: Average Results of the Test Set

accuracy (Table 5.3) in the learning phase. On the other hand, PAT15 using simple linear regression models has worse energy efficiency than the default governor as well as our governor; we observe that mainly GPU-dominant applications such as Dreambike, Epicitadel and Anomaly2 series (the first category in Figure 5.11) result in substantially worse EpF than the default because the prediction errors of S.L.R by ΔF_G (Table 5.3) are significantly higher than those of S.L.R by ΔF_C.

Unlike GPU-dominant applications, most CPU-dominant applications for PAT15 achieved better EpF compared to the default (the second category in Figure 5.11). According to our observations, energy efficiency for mobile gaming benchmarks are mainly dependent on characteristics of benchmarks and platform characters such as CPU/GPU frequency levels and min/max frequency. (Note that the default CPU governor supports cluster-based interactive DVFS policy with nine frequency levels (from 1.2 to 2.0Ghz) in CPU-bc while the GPU governor supports six frequency levels from very low to high frequency (from 177 to 543Mhz)). From the observation that prediction errors of models to GPU frequency are higher than those to CPU frequency (Table 5.3), we speculate the reasons that all GPU cores are only dedicated for rendering tasks while CPU runs a lot of background tasks as well as the graphics rendering task running one of four cores and that response variables change sharply for GPU-dominant applications because the ODROID-XU3 integrated GPU has small number



(a) Energy per Frame



(b) FPS

Figure 5.11: Results of the Test Set (Detailed)

of GPU frequency levels between min and max frequency (e.g., Intel MinnowBoard MAX integrated GPU has nine frequencies ranging from 200 to 511Mhz [40], compared to our six levels ranging from 177 to 543Mhz).

When GPU-workloads are mixed additionally onto CPU-dominant applications using our gaming micro-benchmarks (e.g., mb-101 and mb102 vs. mb111 and mb112: each number stands for CPU, GPU and Memory workloads respectively), overall energy savings are reduced as GPU-workloads add on. Moreover, when memory-workloads are mixed onto CPU-dominant applications (e.g., mb102 and mb112 vs. mb103, mb113 and robocop game), all governors have EpF improvements without FPS degradation. This is because all com-

pared governors except the default (only utilization-based policy) are using target-FPS based policy. In other words, without a specific model that is aware of memory-workloads, a target-FPS based policy can improve energy savings for memory intensive CPU-dominant applications because QoS-based governors can repeatedly reduce the CPU frequency within the target-FPS.

5.4.3 Discussion

In this chapter, we compared the results of two methodologies: HiCAP (a heuristic modeling based methodology described in Ch.4) and ML-Gov (a machine learning enhanced prediction model based methodology described in Ch.5). Here, we discuss the lessons learned from using offline M.L approaches targeting embedded systems in terms of strength, weakness, opportunity and threat (SWOT), compared to heuristic or traditional statistical approaches.

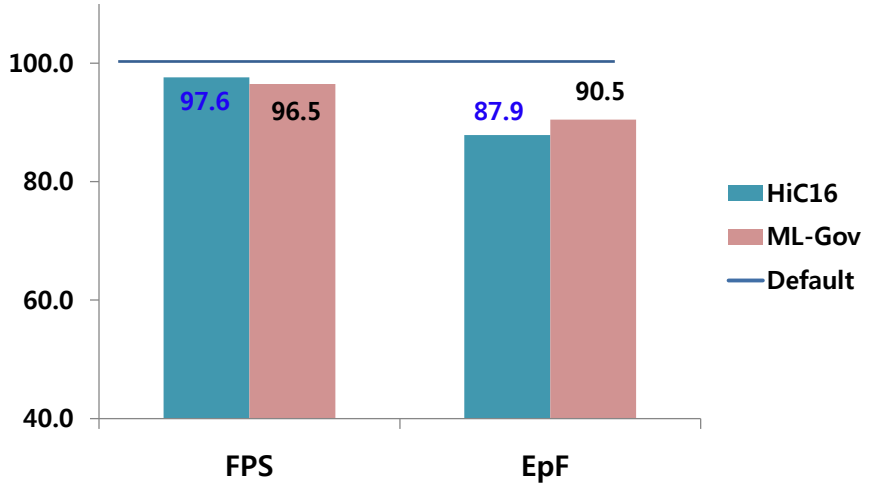


Figure 5.12: Results Comparison between HiCAP and ML-Gov

Compared to HiCAP, on average ML-Gov has almost similar FPS but a little bit worse EpF as shown in Figure 5.12. We observe that ML-Gov has high prediction errors in the GPU models compared to the CPU models (e.g., 3.1% vs. 0.8% in the prediction errors for ΔQ by ΔF_G and ΔF_C respectively) and that ML-Gov framework still has non-trivial reference values such as $\hat{Q}$, $\hat{U_C}$ and $\hat{U_G}$, while HiCAP uses a well-defined heuristic HFSM-

model and different policy in each leaf state on top of effective reference tables (i.e., the CPU-GPU saturated frequency LUTs) using a large amount of benchmarks.

Strengths: Compared to classical statistical models such as simple or multivariate linear models, M.L approaches can build prediction models with high accuracy and simple structure of cost functions using a data mining tool. Furthermore, many non-trivial variables or numeric thresholds can be chosen effectively by already qualified M.L algorithms.

Weaknesses: The model building methodology/process could be complicated because we need to consider characteristics or constraints of a target system from the learning phase; and model building techniques are dependent on algorithms or options built in a data mining tool.

Opportunities: In addition to the *Weka* [41] data mining tool, many usable and practical machine learning tools are emerging and providing already qualified useful algorithms and visualizations with powerful analysis capability. M.L approaches are facilitated by the ability to collect large amounts of data from mobile systems, allowing the exploitation of the diverse and dynamic workload characteristics of emerging heterogeneous embedded systems.

Threats: Design concepts using M.L approaches for multi-input, multi-output and multi-objective optimizations on HMPSoCs are still very challenging. Even if we build perfect models in terms of accuracy and structure of cost functions in the learning phase, the prediction phase may have non-trivial values (e.g., thresholds or reference values) related to a system; sometimes, these kinds of values are more significant than the predicted values by the built models in terms of overall performance improvements of the system.

5.5 Conclusion

In this chapter, we proposed *ML-Gov: A Machine Learning Enhanced Integrated CPU-GPU DVFS Governor for Mobile Gaming*. ML-Gov exploits model building through offline ma-

chine learning techniques; and uses an integrated CPU-GPU DVFS methodology estimating energy-efficient frequencies with the models during runtime. For the learning phase, we collected training data using a set of 20 mobile games exhibiting diverse and dynamic characteristics; we performed attribute selection to remove unrelated variables to a response variable and reduce the complexity of structure of cost functions; and we built tree-based piecewise regression models using machine learning techniques built in the data mining tool. For the prediction phase, we developed a heuristic Hierarchical FSM-based governor framework considering QoS and CPU/GPU bottlenecks; we then set the CPU and GPU frequencies at run-time using the built models as outputs of the HFSM state transitions. Our experimental results on the ODROID-XU3 platform across a test set of 20 mobile games show that our governor achieved significant energy efficiency gains of over 10% improvement in energy-per-frame over a state-of-the-art governor which built simple linear regression models, with a surprising 3% improvement in FPS performance. We believe ML-Gov presents a practical machine learning enhanced method to build models from dynamic data at numerous different hardware configurations on dynamic applications (workloads) of HMPSoC platforms. The work presented here uses offline machine learning techniques for online estimation, with future work addressing: online machine learning methodology and implementation for integrated CPU-GPU DVFS governor. While our ML-Gov methodology was targeted mainly for mobile games, we believe it can also be applicable for various other classes of CPU-GPU integrated graphics applications. Finally, we compared the results of HiCAP and ML-Gov, and discussed advantages and disadvantages of machine learning approaches targeting embedded systems.

Chapter 6

Conclusion and Future Directions

6.1 Summary

The increasing use of mobile platforms for 3D games and other graphics-intensive applications has resulted in deployment of high-performance Heterogeneous MultiProcessor Systems-on-Chip (HMPSoC) with integrated GPUs. However, high performance mobile HMPSoCs result in high power consumption in CPU and GPU with more dynamism. In order to achieve high performance with energy-efficiency for heterogeneous CPU-GPU based architectures that execute mobile games and other graphics-intensive applications, contemporary mobile platforms use software governors which employ Dynamic Voltage Frequency Scaling (DVFS) techniques.

Through comprehensive observations from CPU and GPU DVFS studies for gaming workloads on mobile heterogeneous platforms, we list our motivation as follows: 1) There have been no previous systematic studies to correlate the performance, power, and energy efficiency of mobile GPUs based on diverse graphics workloads to enable more efficient mobile platform DVFS policies for energy savings. 2) Traditionally, separate CPU and GPU gov-

ernors are deployed in order to achieve energy efficiency with high performance through DVFS, but miss opportunities for further energy savings through coordinated system-level application of DVFS (i.e., frequency capping). 3) Mobile games typically exhibit inherent behavioral dynamism, which existing governor policies are unable to exploit effectively to manage CPU/GPU DVFS policies. 4) For dynamic and diverse gaming workloads, existing governors utilize statistical or heuristic models with a small set of mobile games for both modeling and evaluation resulting in high prediction errors in modeling, and do not exploit practical machine learning approaches for prediction models with high accuracy and low complexity using a large amount of various training data.

However, in terms of cooperative design concepts of CPU-GPU dynamic power management (DPM) for battery-based commercial mobile platforms, a challenging goal is that power and performance issues should be considered simultaneously. In addition to this challenging goal, there are a few more specific challenging issues: 1) mobile graphics workloads (especially gaming workloads) are highly dynamic and diverse, requiring graphics workloads characterization for dynamic power management design. 2) mobile platforms are changing rapidly, therefore a simple and easily portable methodology should be developed. 3) cooperative design concepts for CPU-GPU DPM are very complicated, requiring more effective and practical modeling techniques.

6.2 Contributions

To address the challenges outlined earlier, the key contributions of this thesis are listed as follows:

- Graphics Workload Characterization: Comprehensive observations and thorough analyses from the results of micro-benchmarks provide the correlation between workloads of hardware pipeline stages and performance/power effects which provide us opportunities

for energy-efficient mobile DVFS design based on the analyses. Unlike other related work, for comprehensiveness and completeness, this work uses large sets of games (over 100 real games and a few hundred custom micro-benchmarks) and uses an automatic measurement tool. Then, combinations of the custom micro-benchmarks and the result data are extensively used for workload characterization, workload analysis, model building (learning) and evaluations

- Simple but highly effective CPU-GPU Frequency Capping: For rapidly changing mobile platforms, a cooperative frequency capping governor is proposed to achieve energy efficiency for a diverse set of mobile games by building simple and easily portable CPU-GPU Lookup Tables.

- Hierarchical FSM-based Dynamic Behavior Modeling: For effective adaptation of dynamic behavior changes, we propose a Hierarchical FSM (HFSM) based dynamic behavior modeling strategy for mobile gaming and present a cooperative CPU-GPU governor that deploys a simple maximum frequency-capping methodology exploiting the HFSM for dynamic DVFS.

- Machine Learning enhanced Simple and Accurate Prediction Models: To build simple and accurate prediction models for diverse and dynamic gaming workloads on heterogeneous mobile platforms, we propose machine learning enhanced performance models by building tree-based piecewise linear regression models using off-line machine learning algorithms built in a data mining tool. We then present an integrated CPU-GPU DVFS governor that applies piecewise policies with analyses of the models for energy-efficiency with minimal performance degradation.

6.3 Future Directions

As future directions, there could be two big branches in M.L approaches for embedded systems: 1) Offline M.L approach with more M.L-suitable design and 2) Practical and effective online (or hybrid) M.L approach for Embedded Systems.

- Offline M.L approach with more M.L-suitable design: For example, a new governor framework design for direct CPU-GPU frequency settings by building regression tree based LUTs reducing/removing non-trivial values of the prediction phase could be available. The challenging issue of design concepts using M.L approaches on HMP-SoCs is that a model should consider multi-input, multi-output and multi-objective optimizations. If we can reduce non-trivial values of the prediction phase by designing one objective function satisfying the multi-objectives, it is possible to build regression tree based LUTs for multi-outputs satisfying (as much as possible) the objective function by using offline M.L techniques.

- Practical and effective online (or hybrid) M.L approach: Even though online M.L approach is an attractive methodology, according to our observations, the effectiveness for online learning should be considered very carefully for diverse and dynamic workloads due to high variations (fluctuations) of non-trivial coefficients (e.g., online learning based parameter update). Therefore, a practical and effective online (or hybrid) M.L approach is needed for diverse and dynamic mobile gaming workloads. For example, a hybrid (offline + online) methodology could be an alternative for diverse and dynamic mobile gaming. In other words, first using offline M.L techniques, characterization of practical clustering or classification can be developed using collected datasets. Then, by using an online M.L methodology (e.g., parameter update, incremental decision tree generation or single layer neural network), we can build online models piecewisely with a small number of clustering or classification.

Bibliography

[1] Sysfs - the filesystem for exporting kernel objects. https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt.

[2] Tiled rendering. https://en.wikipedia.org/wiki/Tiled_rendering.

[3] A. Acquaviva, L. Benini, and B. Ricco. An adaptive algorithm for low-power streaming multimedia processing. In *Design, Automation and Test in Europe (DATE)*, 2001.

[4] D. Aha and D. Kibler. Instance-based learning algorithms. In *Machine Learning*, 1991.

[5] Anandtech. Qualcomm Snapdragon S4 Performance Preview - MSM8960 MDP and Adreno 225 Benchmarks. http://www.anandtech.com/show/5559/qualcomm-snapdragon-s4-krait-performance-preview-msm8960-adreno-225-benchmarks/4, 2012.

[6] Anandtech. Qualcomm's quad-core snapdragon s4 (apq8064/adreno 320) performance preview. http://www.anandtech.com/show/6112/qualcomms-quadcore-snapdragon-s4-apq8064adreno-320-performance-preview, July 2012.

[7] Anandtech. Snapdragon 800 (MSM8974) Performance Preview. http://www.anandtech.com/show/7082/snapdragon-800-msm8974-performance-preview-qualcomm-mobile-development-tablet, 2014.

[8] Android. monkeyrunner. http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[9] Android. SurfaceFlinger and Hardware Composer. https://source.android.com/devices/graphics/arch-sf-hwc#surfaceflinger.

[10] Apple. App Store. https://www.apple.com/ios/app-store/.

[11] Apple. iPhone. https://www.apple.com/iphone/.

[12] ARM. ARM's GPU roadmap; the WHY. https://community.arm.com/graphics/b/blog/posts/arm-s-gpu-roadmap-the-why, 2013.

[13] ARM. Open source mali midgard gpu kernel drivers. https://developer.arm.com/products/software/mali-drivers/midgard-kernel, Apr. 2014.

[14] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis. Parallel frame rendering: trading responsiveness for energy on a mobile gpu. In *PACT '13 Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 83–92, Oct. 2013.

[15] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis. Teapot: a toolset for evaluating performance, power and image quality on mobile graphics systems. In *ICS '13 Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 37–46, June 2013.

[16] Y. Bai and P. Vaidya. Memory characterization to analyze and predict multimedia performance and power in embedded systems. In *ICASSP*, 2009.

[17] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *Proc. Usenix Ann. Tech. Conf. (UsenixATC)*, pages 21–34, 2010.

[18] Z. Cheng, X. Li, B. Sun, J. Song, C. Wang, and X. Zhou. Behavior-aware integrated cpu-gpu power management for mobile games. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Sept. 2016.

[19] K. Choi, K. Dantu, W.-C. Cheng, and M. Pedram. Frame-based dynamic voltage and frequency scaling for a mpeg decoder. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, 2002.

[20] P.-K. Chuan, Y.-S. Chen, and P.-H. Huang. An adaptive on-line cpu-gpu governor for games on mobile devices. In *Design Automation Conference (ASP-DAC)*, Jan. 2017.

[21] T. cker Chiueh and W. jen Lin. Characterization of static 3d graphics workloads. In *Proceedings of the 1997 SIGGRAPH/Eurographics workshop on Graphics hardware*, pages 17–24, 1997.

[22] B. Dietrich and S. Chakraborty. Managing power for closed-source android os games by lightweight graphics instrumentation. In *Network and Systems Support for Games (NetGames), 2012 11th Annual Workshop*, pages 1–3, Nov. 2012.

[23] B. Dietrich and S. Chakraborty. Power management using game state detection on android smartphones. In *MobiSys '13 Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 493–494, June 2013.

[24] B. Dietrich and S. Chakraborty. Forget the battery, lets play games! In *ESTIMedia*, 2014.

[25] B. Dietrich and S. Chakraborty. Lightweight graphics instrumentation for game state-specific power management in android. In *Multimedia Systems*, 2014.

[26] B. Dietrich, D. Goswami, S. Chakraborty, A. Guha, and M. Gries. Lms-based low-complexity game workload prediction for dvfs. In *ICCD*, pages 417–424, 2010.

[27] T. Dwyer, A. Fedorova, S. Blagodurov, M. Roth, F. Gaud, and J. Pei. A practical method for estimating performance degradation on multicore processors, and its application to hpc workloads. In *the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

[28] F. Ercan, N. A. Gazala, and H. David. An integrated approach to system-level cpu and memory energy efficiency on computing systems. In *Energy Aware Computing*, pages 1–6, 2012.

[29] Extremetech. Apples A8 SoC analyzed. `https://www.extremetech.com/computing/189787-apples-a8-soc-analyzed-the-iphone-6-chip-is-a-2-billion-transistor-20nm-monster`, 2014.

[30] R. Ge, R. Vogt, J. Majumder, A. Alam, M. Burtscher, and Z. Zong. Effects of dynamic voltage and frequency scaling on a k20 gpu. In *Parallel Processing (ICPP), 2013 42nd International Conference*, pages 826 – 833, Oct. 2013.

[31] A. Girault, B. Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.*, 18(6):742–760, 1999.

[32] Google. Android. `https://www.android.com/`.

[33] Google. Google Play Store. `https://play.google.com/store`.

[34] Google. Butter project. `https://developers.google.com/events/io/2012/sessions/gooio2012/109/`, 2012.

[35] P. Greenhalgh. Big.little processing with arm cortex-a15 and cortex-a7. In *An ARM White paper*, 2011.

[36] Y. Gu and S. Chakraborty. Control theory-based dvs for interactive 3d games. In *DAC*, pages 740–745, 2008.

[37] Y. Gu and S. Chakraborty. A hybrid dvs scheme for interactive 3d games. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 3–12, 2008.

[38] Y. Gu and S. Chakraborty. Power management of interactive 3d games using frame structure information. In *VLSI Design, 2008. VLSID 2008. 21st International Conference*, pages 679 – 684, Jan. 2008.

[39] Y. Gu, S. Chakraborty, and W. T. Ooi. Games are up for dvfs. In *DAC '06 Proceedings of the 43rd annual Design Automation Conference*, pages 598–603, July 2006.

[40] U. Gupta, J. Campbell, R. Ayoub, M. Kishinevsky, and S. Gumussoy. Adaptive performance prediction for integrated gpus. In *ICCAD*, 2016.

[41] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, June 2009.

[42] M. A. Hall. *Correlation-based Feature Subset Selection for Machine Learning.* PhD thesis, University of Waikato, Hamilton, New Zealand, 1998.

[43] Hardkernel. Odroid-xu3. `http://odroid.com/dokuwiki/doku.php?id=en:odroid-xu3`, Aug. 2015.

[44] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program*, 8:231–274, 1987.

[45] J. Hasselgren and T. Akenine-Moller. An efficient multi-view rasterization architecture. In *Proceedings of the 17th Eurographics conference on Rendering Techniques, ser. (EGSR)*, pages 61–72, 2006.

[46] C.-Y. Hsieh, J.-G. Park, N. Dutt, and S.-S. Lim. Memory-aware cooperative cpu-gpu dvfs governor for mobile games. In *Embedded Systems For Real-time Multimedia (ESTIMedia)*, 2015.

[47] C. J. Hughes and S. V. Adve. A formal approach to frequent energy adaptations for multimedia applications. In *International Symposium on Computer Architecture (ISCA)*, pages 138–149, 2004.

[48] C. Im, S. Ha, and H. Kim. Dynamic voltage scheduling with buffers in low-power multimedia applications. In *ACM Transactions in Embedded Computing Systems*, pages 686–705, 2004.

[49] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver. A qos-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc. In *Proceedings of the 49th Annual Design Automation Conference (DAC)*, pages 850–855, 2012.

[50] H. Jung, C. Lee, S. haeng Kang, S. Kim, H. Oh, and S. Ha. Dynamic behavior specification and dynamic mapping for real-time embedded systems: Hopes approach. *ACM Trans. Embedd. Comput. Syst.*, 13(4):135–161, March 2014.

[51] D. Kadjo, R. Ayoub, M. Kishinevsky, and P. V. Gratz. A control-theoretic approach for energy efficient cpu-gpu subsystem in mobile systems. In *DAC*, 2015.

[52] D. Kadjo, U. Ogras, R. Ayoub, M. Kishinevsky, and P. Gratz. Towards platform level power management in mobile systems. In *System-on-Chip Conference (SOCC)*, pages 146–151, 2014.

[53] Khronos Group. Opengl es. `https://www.khronos.org/opengles/`.

[54] Y. G. Kim, M. Kim, J. M. Kim, M. Sung, and S. W. Chung. A novel gpu power model for accurate smartphone power breakdown. In *ETRI Journal*, pages 157–164, 2015.

[55] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. Gpuwattch: Enabling energy optimizations in gpgpus. In *Proc of the 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 487–498, 2013.

[56] Letustweak. Snapdragon 820 Features and Performance. http://www.letustweak.com/tweaks/qualcomm-snapdragon-820-features-performa, 2016.

[57] LG. Nexus 4 e960. http://www.lg.com/uk/mobile-phones/lg-E960-nexus-4-by-lg, Jan. 2013.

[58] X. Li, G. Yan, Y. Han, and X. Li. Smartcap: User experience-oriented power adaptation for smartphone's application processor. In *DATE*, 2013.

[59] W.-Y. Liang, Y.-L. Chen, and M.-F. Chang. A memory-aware energy saving algorithm with performance consideration for battery-enabled embedded systems. In *International Symposium on Consumer Electronics (ISCE)*, pages 547–551, 2011.

[60] Linux(TM). Linux cpufreq governors (new 'interactive' governor). https://android.googlesource.com/kernel/common/+/android-4.4/Documentation/cpufreq/governors.txt, June 2010.

[61] M. L. Loper. *Modeling and Simulation in the Systems Engineering Life Cycle*. Springer-Verlag, London, pp. 75-81, 2015.

[62] Z. Lu, J. Hein, M. Humphrey, M. Stan, J. Lach, and K. Skadron. Control-theoretic dynamic frequency and voltage scaling for multimedia workloads. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 156–163, New York, NY, USA, 2002.

[63] Z. Lu, J. Lach, M. Stan, and K. Skadron. Reducing multimedia decode power using feedback control. In *International Conference on Computer Design (ICCD)*, pages 489–497, 2003.

[64] X. Ma, Z. Deng, M. Dong, and L. Zhong. Characterizing the performance and power consumption of 3d mobile games. In *Computer (Volume:46 , Issue: 4)*, pages 76 – 82, 2013.

[65] X. Ma, M. Dong, L. Zhong, and Z. Deng. Statistical power consumption analysis and modeling for gpu-based computing. In *HotPower '09 Proceedings of the Workshop on Power-Aware Computing and Systems*, Nov. 2009.

[66] A. Maghazeh, U. D. Bordoloi, P. Eles, and Z. Peng. General purpose computing on low-power embedded gpus: Has it come of age? In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2013.

[67] A. Maghazeh, U. D. Bordoloi, M. Villani, P. Eles, and Z. Peng. Perception-aware power management for mobile games via dynamic resolution scaling arian. In *International Conference on Computer-Aided Design (ICCAD)*, 2015.

[68] A. Mallik, B. Lin, G. Memik, P. Dinda, and R. P. Dick. User-driven frequency scaling. In *IEEE Computer Architecture Letters*, page 5(2):16, 2006.

[69] P. Marwedel. *Embedded System Design*. Springer, B.V, pp. 42-52, 2011.

[70] X. Mei, L. S. Yung, K. Zhao, and X. Chu. A measurement study of gpu dvfs on energy conservation. In *HotPower '13 Proceedings of the Workshop on Power-Aware Computing and Systems*, page Article No. 10, Nov. 2013.

[71] Metaps. US 2014 Analysis. `http://www.metaps.com/press/en/blog/161-ustrends1209`, 2014.

[72] T. Mitra and T. Chiueh. Dynamic 3d graphics workload characterization and the architectural implications. In *Proc. Int'l Symp. Microarchitectures (Micro-32)*, pages 62–71, 1999.

[73] B. Mochocki, K. Lahiri, and S. Cadambi. Power analysis of mobile 3d graphics. In *Proc. Conf. Design, Automation and Test in Europe (DATE)*, pages 502–507, 2006.

[74] Monsoon Solutions Inc. Monsoon Power Monitor. `http://www.msoon.com/LabEquipment/PowerMonitor/`, 2008.

[75] J.-G. Park, C.-Y. Hsieh, N. Dutt, and S.-S. Lim. Quality-aware mobile graphics workload characterization for energy-efficient DVFS design. In *IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, Oct. 2014.

[76] J.-G. Park, C.-Y. Hsieh, N. Dutt, and S.-S. Lim. Cooperative CPU-GPU frequency capping (Co-Cap) for energy efficient mobile gaming. In *UCI Center for Embedded and Cyber-physical Systems TR*, 2015.

[77] J.-G. Park, C.-Y. Hsieh, N. Dutt, and S.-S. Lim. Co-cap: Energy-efficient cooperative cpu-gpu frequency capping for mobile games. In *SAC*, 2016.

[78] J.-G. Park, H. Kim, N. Dutt, and S.-S. Lim. Hicap: Hierarchical fsm-based dynamic integrated cpu-gpu frequency capping governor for energy-efficient mobile gaming. In *ISLPED*, Aug. 2016.

[79] J.-G. Park, H. Kim, N. Dutt, and S.-S. Lim. Using hsfms to model mobile gaming behavior for energy efficient dvfs governors. In *UCI Center for Embedded and Cyber-physical Systems TR*, 2016.

[80] A. Pathania, A. E. Irimiea, A. Prakash, and T. Mitra. Power-performance modelling of mobile gaming workloads on heterogeneous MPSoCs. In *DAC*, 2015.

[81] A. Pathania, Q. Jiao, A. Prakash, and T. Mitra. Integrated cpu-gpu power management for 3d mobile games. In *DAC '14 Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, pages 1–6, June 2014.

[82] A. Pathania, Q. Jiao, A. Prakash, and T. Mitra. Integrated CPU-GPU power management for 3D mobile games. In *DAC*, June 2014.

[83] I. Paul, V. Ravi, S. Manne, M. Arora, and S. Yalamanchili. Coordinated energy management in heterogeneous processors. In *Scientific Programming*, pages 22: 93–108, 2014.

[84] A. Prakash, H. Amrouch, M. Shafique, T. Mitra, and J. Henkel. Improving mobile gaming performance through cooperative cpu-gpu thermal management. In *Design Automation Conference (DAC)*, June 2016.

[85] Qualcomm. Adreno Profiler. `https://developer.qualcomm.com/blog/adreno-profiler-gets-close-and-personal-gpu`, 2011.

[86] R. J. Quinlan. Learning with continuous classes. In *Australian Joint Conference on Artificial Intelligence*, 1992.

[87] S. Rabin. *GAME AI PRO*. CRC Press, LLC, pp. 47-52, 2014.

[88] J. Roca, V. M. D. Barrio, C. Gonzalez, and C. Solis. Workload characterization of 3d games. In *IISWC*, pages 17–26, 2006.

[89] Samsung. Exynos 5 octa (5422). `http://www.samsung.com/semiconductor/minisite/Exynos/w/solution/mobile_ap/5422/`.

[90] R. Schone, D. Hackenberg, and D. Molka. Memory performance at reduced cpu clock speeds: an analysis of current x86_64 processors. In *HotPower'12 Proceedings of the 2012 USENIX conference on Power-Aware*, pages 5–9, Oct. 2012.

[91] J. Sheaffer, D. Luebke, and K. Skadron. A flexible simulation framework for graphics architectures. In *Proc. ACM SIGGRAPH/Eurographics Conf. Graphics Hardware (GH)*, pages 85–94, 2004.

[92] Statista. Number of available applications in the Google Play Store from December 2009 to June 2017. `https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/`, 2017.

[93] Texas Instruments. Ina231. `http://www.ti.com/product/INA231`.

[94] H. Wang, V. Sathish, R. Singh, M. J. Schulte, , and N. S. Kim. Workload and power budget partitioning for singlechip heterogeneous processors. In *arallel architectures and compilation techniques (PACT)*, pages 401–410, 2012.

[95] Y. Wang and I. H. Witten. Induction of model trees for predicting continuous classes. In *European Conference on Machine Learning*, 1997.

[96] Wikipedia. Smartphone. `https://en.wikipedia.org/wiki/Smartphone`, 2017.

[97] C. Willmott and K. Matsuura. Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance. *Clim. Res.*, 30:79–82, Dec. 2005.

[98] M. Wimmer and P. Wonka. Rendering time estimation for real-time rendering. In *Proc. Eurographics Workshop Rendering (EGWR)*, pages 118–129, 2003.

[99] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou. Gpgpu performance and power estimation using machine learning. In *HPCA*, 2015.

[100] D. You and K.-S. Chung. Dynamic voltage and frequency scaling framework for low-power embedded gpus. In *Electronics Letters*, pages 48(21):1333–1334, 2012.

[101] D. You and K.-S. Chung. Quality of service-aware dynamic voltage and frequency scaling for embedded gpus. In *IEEE Computer Architecture Letter*, pages 14(1):66–69, 2014.

[102] W. Yuan and K. Nahrstedt. Practical voltage scaling for mobile multimedia devices. In *Proceedings of the 12th Annual ACM International Conference on Multimedia*, pages 924–931, New York, NY, USA, 2004.